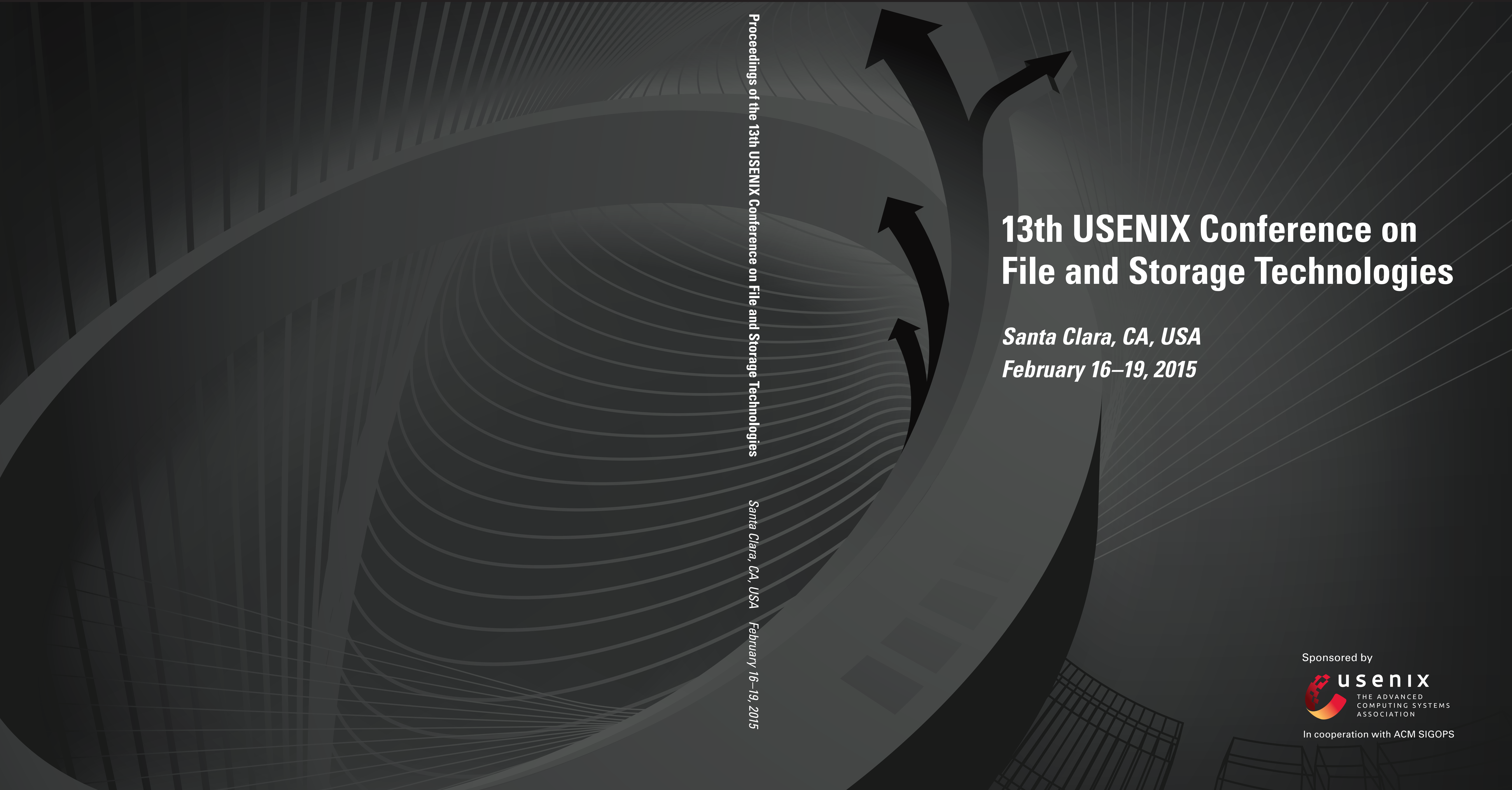
usenix
ASSOCIATION

Proceedings of the 13th USENIX Conference on File and Storage Technologies

Santa Clara, CA, USA   February 16–19, 2015

conference

proceedings



13th USENIX Conference on File and Storage Technologies

Santa Clara, CA, USA

February 16–19, 2015

Sponsored by

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

In cooperation with ACM SIGOPS

Thanks to Our FAST '15 Sponsors

Platinum Sponsor

NetApp

Gold Sponsors

EMC² Google NSF vmware

Bronze Sponsors

facebook hp IBM Research Microsoft Research

pernixdata SNIA

General Sponsors

BERKELEY COMMUNICATIONS NUTANIX

Media Sponsors and Industry Partners

ACM *Queue*
ADMIN magazine
CRC Press
Datanami
Distributed Management Task Force (DMTF)
EnterpriseTech
FreeBSD Foundation
HPCwire
InfoSec News
Linux Foundation
Linux Journal
Linux Pro Magazine
No Starch Press
Raspberry Pi Geek
UserFriendly.org

ISBN 978-1-931971-201

Thanks to Our USENIX and LISA SIG Supporters

USENIX Patrons

Facebook Google Microsoft Research
NetApp VMware

USENIX Benefactors

Akamai Hewlett-Packard
IBM Research *Linux Pro Magazine*

USENIX and LISA SIG Partners

Booking.com Cambridge Computer
Can Stock Photo Fotosearch Google

USENIX Partners

EMC Huawei Meraki

USENIX Association

Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST ’15)

February 16–19, 2015
Santa Clara, CA, USA

Conference Organizers

Program Co-Chairs
Jiri Schindler, *SimpliVity*
Erez Zadok, *Stony Brook University*

Program Committee
Nitin Agrawal, *NEC Labs*
Ahmed Amer, *Santa Clara University*
Angela Demke Brown, *University of Toronto*
Randal Burns, *Johns Hopkins University*
Peter Desnoyers, *Northeastern University*
Fred Douglis, *EMC*
Michael Factor, *IBM Research—Haifa*
Ashvin Goel, *University of Toronto*
Haryadi Gunawi, *University of Chicago*
Dean Hildebrand, *IBM Research—Almaden*
Cheng Huang, *Microsoft Research*
Stratos Idreos, *Harvard University*
Kimberly Keeton, *HP Labs*
Geoff Kuenning, *Harvey Mudd College*
Darrell Long, *University of California, Santa Cruz*
Xiaosong Ma, *Qatar Computing Research Institute and North Carolina State University*
Arif Merchant, *Google*
Jason Nieh, *Columbia University*
Sam H. Noh, *Hongik University*
John Ousterhout, *Stanford University*
Donald Porter, *Stony Brook University*
Raju Rangaswami, *Florida International University*
Philip Shilane, *EMC*
Liuba Shrira, *Brandeis University*
John Strunk, *NetApp*
Nisha Talagala, *SanDisk*
Jens Teubner, *TU Dortmund University*
Theodore Ts'o, *Google*
Carl Waldspurger, *CloudPhysics*
Youjip Won, *Hanyang University*

Steering Committee
Remzi Arpaci-Dusseau, *University of Wisconsin—Madison*
William J. Bolosky, *Microsoft Research*
Randal Burns, *Johns Hopkins University*
Jason Flinn, *University of Michigan*
Greg Ganger, *Carnegie Mellon University*
Garth Gibson, *Carnegie Mellon University and Panasas*
Casey Henderson, *USENIX Association*
Kimberly Keeton, *HP Labs*
Erik Riedel, *EMC*
Bianca Schroeder, *University of Toronto*
Margo Seltzer, *Harvard School of Engineering and Applied Sciences and Oracle*
Keith A. Smith, *NetApp*
Eno Thereska, *Microsoft Research*
Ric Wheeler, *Red Hat*
Yuanyuan Zhou, *University of California, San Diego*

Tutorial Coordinator
John Strunk, *NetApp*

External Reviewers

Kavita Agarwal
Amogh Akshintala
Jongmoo Choi
Bhushan Jain
William Jannen
Dongwoo Kang
Phil Keller
Jaeho Kim
Raghav Lagisetty
Eunji Lee
Cheng Li
Youyou Lu
Fei Meng
Ethan L. Miller
Ankur Mittal
Yongseok Oh
Phaneendra Reddy
Mahesh Sathiamoorthy
Alexander Shraer
Jay Wylie
Yang Zhan
Tao Zhang
Zhao Zhang

13th USENIX Conference on File and Storage Technologies (FAST '15) February 16–19, 2015 Santa Clara, CA, USA

Wednesday, February 18, 2015

Thursday, February 19, 2015

Message from the FAST ’15 Program Co-Chairs

Welcome to the 13th USENIX Conference on File and Storage Technologies. This year’s conference continues the FAST tradition of bringing together researchers and practitioners from both industry and academia for a program of innovative and rigorous storage-related research. We are pleased to present a diverse set of papers on topics such as scaling for big data and distributed systems, erasure codes, SSD and SMR, reliability and performance, write-optimized systems, benchmarking and workloads, and mobile and social-networking systems. Our authors hail from many countries on three continents and represent academia, industry, and the open-source communities. Many of the submitted papers are the fruits of a collaboration among all these communities.

FAST ’15 received 130 submissions. Of these, we selected 28, for an acceptance rate of 21%. The Program Committee used a two-round online review process, and then met in person to select the final program. In the first round, each paper received at least three reviews. For the second round, 68 papers received at least two more reviews. The Program Committee discussed 54 papers in an all-day meeting on December 5, 2014, at Stony Brook University, New York, USA. We used Eddie Kohler’s superb HotCRP software to manage all stages of the review process, from submission to author notification.

As in the previous three years, we have included a category of short papers in the program. Short papers provide a vehicle for presenting research ideas that do not require a full-length paper to describe and evaluate. In judging short papers, we applied the same standards as for full-length submissions. 25 of our submissions were short papers, of which we accepted four. This year, we also included an option to indicate an asset release with the submission—be it source code, traces, or other artifacts—that others in the FAST community can use and benefit from; 28 submissions selected this option. Finally, we were happy to see a growing number of submissions (and accepted papers) from adjacent areas such as database systems and hardware.

We wish to thank the many people who contributed to this conference. First and foremost, we are grateful to all the authors who submitted their work to FAST ’15. We would also like to thank the attendees of FAST ’15 and future readers of these papers. Together with the authors, you form the FAST community and make storage research vibrant and exciting. We extend our thanks to the USENIX staff, who have provided outstanding support throughout the planning and organizing of this conference with the highest degree of professionalism and friendliness. Most importantly, their behind-the-scenes work makes this conference actually happen. Our thanks go also to the members of the FAST Steering Committee who provided invaluable advice and feedback.

Finally, we wish to thank our Program Committee for their many hours of hard work in reviewing and discussing the submissions, some of whom traveled half across the world for the one-day in-person PC meeting. Together with a few external reviewers, they wrote over 520 thoughtful and meticulous reviews; in addition, PC members contributed over 480 online comments of discussions during the reviewing rounds. HotCRP recorded over 340,000 words in reviews and comments. The reviewers’ reviews, and their thorough and conscientious deliberations at the PC meeting, contributed significantly to the quality of our decisions. Finally, we also thank several people who helped make the PC meeting run smoothly: student volunteer Ming Chen; the IT staff headed by Ken Gladkey; administrative and local arrangements support from Kathy Germana and Chrisitine Cesaria; and department chair Arie Kaufman for sponsorship.

We look forward to an interesting and enjoyable conference!

Jiri Schindler, *SimpliVity*
Erez Zadok, *Stony Brook University*
FAST ’15 Program Co-Chairs

CalvinFS: Consistent WAN Replication and Scalable Metadata Management for Distributed File Systems

Alexander Thomson
Google[†]
agt@google.com

Daniel J. Abadi
Yale University
dna@cs.yale.edu

Abstract

Existing file systems, even the most scalable systems that store hundreds of petabytes (or more) of data across thousands of machines, store file metadata on a single server or via a shared-disk architecture in order to ensure consistency and validity of the metadata.

This paper describes a completely different approach for the design of replicated, scalable file systems, which leverages a high-throughput distributed database system for metadata management. This results in improved scalability of the metadata layer of the file system, as file metadata can be partitioned (and replicated) across a (shared-nothing) cluster of independent servers, and operations on file metadata transformed into distributed transactions.

In addition, our file system is able to support standard file system semantics—including fully linearizable random writes by concurrent users to arbitrary byte offsets within the same file—across wide geographic areas. Such high performance, fully consistent, geographically distributed files systems do not exist today.

We demonstrate that our approach to file system design can scale to billions of files and handle hundreds of thousands of updates and millions of reads per second—while maintaining consistently low read latencies. Furthermore, such a deployment can survive entire datacenter outages with only small performance hiccups and no loss of availability.

1 Introduction

Today's web-scale applications store and process increasingly vast amounts of data, imposing high scalability requirements on cloud data storage infrastructure.

The most common mechanism for maximizing availability of data storage infrastructure is to replicate all data storage across many commodity machines within a datacenter, and then to keep hot backups of all critical system components on standby, ready to take over in case the main component fails.

However, natural disasters, configuration errors, hunters, and squirrels sometimes render entire datacenters unavailable for spans of time ranging from minutes to days [13, 15]. For applications with stringent availability requirements, replication across multiple geographically separated datacenters is therefore essential.

For certain classes of data storage infrastructure, significant strides have been made in providing vastly scalable solutions that also achieve high availability via WAN replication. For example, replicated block stores, where blocks are opaque, immutable, and entirely independent objects are fairly easy to scale and replicate across datacenters since they generally do not need to support multi-block operations or any kind of locality of access spanning multiple blocks. NoSQL systems such as Cassandra [12], Dynamo [8], and Riak [2] have also managed to achieve both scale and geographical replication, albeit through reduced replica consistency guarantees. Even some database systems, such as the F1 system [19] which Google built on top of Spanner [7] have managed to scalably process SQL queries and ACID transactions while replicating across datacenters.

Unfortunately, file systems have not achieved the same level of scalable, cross-datacenter implementation. While many distributed file systems have been developed to scale to clusters of thousands of machines, these systems do not provide WAN replication in a manner that allows continuous operation in the event of a full datacenter failure due to the difficulties of providing expected file system semantics and tools (linearizable operations, hierarchical access control, standard command-line tools, etc.) across geographical distances.

[†]This work was done while the author was at Yale.

In addition to lacking support for geographical replication, modern file systems—even those known for scalability—utilize a fundamentally unscalable design for metadata management in order to avoid high synchronization costs necessary to maintain traditional file system semantics for file and directory metadata, including hierarchical access control and linearizable writes. Hence, while they are able to store hundreds of petabytes of data (or more) by leveraging replicated block stores to store the *contents* of files, they rely on an assumption that the average file size is very large, while the number of unique files and directories are comparatively small. They therefore run into problems handling large numbers of small files, as the system becomes bottlenecked by the metadata management layer [20, 27].

In particular, most modern distributed file systems use one of two synchronization mechanisms to manage metadata access:

- A special machine dedicated to storing and managing all metadata. GFS, HDFS, Lustre, Gluster, Ursa Minor, Farsite, and XtreemFS are examples of file systems that take this approach [10, 21, 18, 1, 3, 4, 11]. The scalability of such systems are clearly fundamentally bottlenecked by the metadata management layer.
- A shared-disk abstraction that coordinates all concurrent access. File systems that rely on shared disk for synchronization include GPFS, PanFS, and xFS [17, 26, 22]. Such systems generally replicate data across multiple spindles for fault tolerance. However, these typically rely on extremely low (RAID-local or rack-local) synchronization latencies between replicated disks in order to efficiently expose a unified disk address space. Concurrent disk access by multiple clients are synchronized by locking, introducing performance limitations for hot files [17]. Introducing WAN latency synchronization times into lock-hold durations would significantly increase the severity of these limitations.

In this paper, we describe the design of a distributed file system that is substantially different from any of the above-cited file systems. Our system is most distinguished by the metadata management layer which horizontally partitions and replicates file system metadata across a shared-nothing cluster of servers, spanning multiple geographic regions. File system operations that potentially span multiple files or directories are transformed into distributed transactions, and processed via a transaction scheduling and replication management layer of an extensible distributed database system in order to ensure proper coordination of linearizable updates.

Due to the uniqueness of our design, our system, which we call CalvinFS, has a different set of advantages and disadvantages relative to traditional distributed file systems. In particular, our system can handle a nearly unlimited number of files, and can support fully linearizable random writes by concurrent users to arbitrary byte offsets within a file that is consistently replicated across wide geographic areas—neither of which is possible in the above-cited file system designs. However, our system is optimized for operations on single files. Multiple-file operations require distributed transactions, and while our underlying database system can handle such operations at high throughput, the latency of such operations tend to be larger than in traditional distributed file systems.

2 Background: Calvin

As described above, we horizontally partition metadata for our file system across multiple nodes, and file operations that need to atomically edit multiple metadata elements are run as distributed transactions. We extended the Calvin database system to implement our metadata layer, since Calvin has proven to be able to achieve consistent geo-replicated and linear distributed transaction scalability to hundreds of thousands of transactions per second across hundreds of machines per replica, even under relatively high levels of lock contention [24]. The remainder of this section will provide a brief overview of Calvin's architecture and execution protocol.

A Calvin deployment consists of three main components: a transaction request **log**, a **storage layer**, and a **scheduling layer**. Each of these components provides a clean interface and implementations that can be swapped in and out. The log stores a global totally-ordered sequence of transaction requests. Each transaction request in the log represents a read-modify-write operation on the contents of the storage layer; the particular implementation of the storage layer plus any arguments logged with the request define the semantics of the operation. The scheduling layer has the job of orchestrates the (concurrent) execution of logged transaction requests in a manner that is equivalent to a deterministic serial execution in exactly the order they appear in the log.

For each of these three components, we describe here the specific implementation of the component that we used in the metadata subsystem of CalvinFS.

Log

The log implementation we used consists of a large collection of "front-end" servers, an asynchronously- replicated distributed block store, and a small group of "meta-log" servers. Clients append requests to the log by sending them to a front-end server, which batches it with other incoming requests and writes the batch to the distributed block store. Once it is sufficiently replicated in the block store (2 out of 3 replicas have acked the write, say), the front-end server sends the batch's unique block id to a meta- log server. The meta-log servers, which are typically distributed across multiple datacenters, maintain a Paxos-replicated "meta-log" containing a sequence of block ids referencing request batches. The state of the log at any time is considered to be the concatenation of batches in the order specified by the "meta- log".

Storage Layer The storage layer encapsulates all knowledge about physical datastore organization and actual transaction semantics. It consists of a collection of "storage nodes", each of which runs on a different machine in the cluster and maintains a shard of the data. Valid storage layer implementations must include (a) read and write primitives that execute locally at a single node, and (b) a placement manager that determines at which storage nodes these primitives must be run with given input arguments. Compound transaction types may also be defined that combine read/write primitives and arbitrary deterministic application logic. Each transaction request that appears in the log corresponds to a primitive operation or compound transaction. Primitives and transactions may return results to clients upon completion, but their behavior may not depend any inputs other than arguments that are logged with the request and the current state of the underlying data (as determined by read primitives) at execution time.

The storage layer for CalvinFS metadata consists of a multiversion key-value store at each storage node, plus a simple consistent hashing mechanism for determining data placement. The compound transactions implemented by the storage layer are described in Section 5.1.

Scheduler
Each storage node has a local scheduling layer component (called a "scheduler") associated with it which drives local transaction execution.

The scheduling layer takes an unusual approach to pessimistic concurrency control. Traditional database systems typically schedule concurrent transaction execution by checking the safety of each read and write performed by a transaction immediately before that operation occurs, pausing execution as needed (e.g., until an earlier transaction releases an already-held lock on the target record). Each Calvin scheduler, however, examines a transaction before it begins executing at all, decides when it is safe to execute the *whole* transaction based on its read-write sets (which can be discovered automatically or annotated by the client), and then hands the transaction request to the associated storage node, which is free to execute it with no additional oversight.

Calvin's scheduler implementation uses a protocol called deterministic locking, which resembles strict two-phase locking, except that transactions are required to request all locks that they will need in their lifetimes atomically, and in the relative order in which they appear in the log. This protocol is deadlock- free and serializable, and furthermore ensures that execution is equivalent not only to *some* serial order, but to a deterministic serial execution in log order.

All lock management is performed locally by a scheduler, and schedulers track lock requests only for data that resides at the associated storage node (according to the placement manager). When transactions access records spanning multiple machines, Calvin forwards the *entire* transaction request to all schedulers guarding relevant storage nodes. At each participating scheduler, once the transaction has locked all local records, the transaction proceeds to execute using the following protocol:

1. **Perform all local reads.** Read all records in the transaction's read-set that are stored at the local storage node.
2. **Serve remote reads.** Forward each local read result from step 1 to every other participant.
3. **Collect remote read results.** Wait to receive all messages sent by other participants in step 2[1].
4. **Execute transaction to completion.** Once all read results have been received, execute the transaction to completion, applying writes that affect records in the local storage node, and silently dropping writes to data that is not stored locally (since these writes will be applied by other participants).

Upon completion, the transaction's locks are released and the results are sent to the client that originally submitted the transaction request.

A key characteristic of the above protocol is the lack of a distributed commit protocol for distributed transactions. This is a result of the deterministic nature of processing transactions—any failed node can recover its state by loading a recent datat checkpoint and then replaying the log deterministically. Therefore, double checking that no node failed over the course of processing the transaction is unnecessary. The lack of distributed commit protocol, combined with the deadlock-free property of the scheduling algorithm greatly improves the scalability of the system and reduces latency.

[1]If all read results are not received within a specified timeframe, send additional requests to participants to get the results. If there is still no answer, also send requests to other replicas of the unresponsive participant.

2.1 OLLP

Certain file system operations—notably recursive moves, renames, deletes, and permission changes on non-empty directories—were implemented by bundling together many built-in transactions into a single compound transaction. It was not always possible to annotate these compound transaction requests with their full read- and write-sets (as required by Calvin's deterministic scheduler) at the time the recursive operation was initiated. In these cases, we made use of Calvin's Optimistic Lock Location Prediction (OLLP) mechanism [24] as we describe further in Section 5.2.

With OLLP, an additional step is added to the transaction execution pipeline: all transaction requests go through an `Analyze` phase before being appended to the log. The purpose of the `Analyze` phase is to determine the read- and write-sets of the transaction. Stores can implement custom `Analyze` logic for classes of transactions whose read- and write-sets can be statically computed from the arguments supplied by the client, or the `Analyze` function can simply do a "dry run" of the transaction execution, but not apply any writes. In general, this is done at no isolation, and only at a single replica, to make it as inexpensive as possible.

Once the `Analyze` phase is complete, the transaction is appended to the log, and it can then be scheduled and executed to completion. However, it is possible for a transaction's read- and write-sets to grow between the `Analyze` phase and the actual execution (called the `Run` phase) due to changes in the contents of the datastore. In this case, the worker executing the `Run` phase notices that the transaction is attempting to read or write a record that did not appear in its read- or write-set (and which was therefore not locked by the scheduler and cannot be safely accessed). It then aborts the transaction and returns an updated read-/write-set annotation to the client, who may then restart the transaction, this time skipping `Analyze` phase.

3 CalvinFS Architecture

CalvinFS was designed for deployments in which file data and metadata are both (a) replicated with strong consistency across geographically separated datacenters and (b) partitioned across many commodity servers within each datacenter. CalvinFS therefore simultaneously addresses the availability and scalability challenges described above—while providing standard, consistent file system semantics.

We engineered CalvinFS around certain additional goals and design principles:

Main-memory metadata store. Current metadata entries for *all* files and directories must be stored in main-memory across a shared-nothing cluster of machines.

Potentially many small files. The system must handle billions distinct files.

Scalable read/write throughput. Read and write *throughput* capacity must scale near-linearly and must not depend on replication configuration.

Tolerating slow writes. High update latencies that accommodate WAN round trips for the purposes of consistent replication are acceptable.

Linearizable and snapshot reads. When reading a file, clients must be able to specify one of three modes, each with different latency costs:

- **Full linearizable read.** If a client requires fully linearizable read semantics when reading a file, the read may be required to go through the same log-ordering process as any update operation.
- **Very recent snapshot read.** For many clients, very low-latency reads of extremely recent file system snapshots are preferable to higher-latency linearizable reads. We specifically optimize CalvinFS for this type of read operation, allowing for up to 400ms of staleness. (Note that this only applies to read-only operations. Read-modify-write operations on metadata—such as permissions checks before writing to a file—are always linearizable.)
- **Client-specified snapshot read.** Clients can also specify explicit version/timestamp bounds on snapshot reads. For example, a client may choose to limit staleness to make sure that a recent write is reflected in a new read, even if this requires blocking until all earlier writes are applied at the replica at which the read is occurring. Or a client may choose to perform snapshot read operations at a historical timestamp for the purposes of auditing, restoring a backup, or other historical analyses. Since only current metadata entries for each file/directory are pinned in memory at all times, it is acceptable for historical snapshot reads to incur additional latency when digging up now-defunct versions of metadata entries.

Hash-partitioned metadata. Hash partitioning of file metadata based on full file path is preferable to range- or subtree-partitioning, because it typically provides better load balancing and simplifies data placement tracking. Nonetheless, identifying the contents of a directory should only require reading from a single metadata shard.

Optimize for single-file operations. The system should be optimized for operations that create, delete, modify, or read one file or directory at a time[2]. Recursive metadata operations such as directory copies, moves, deletes, and owner/permission changes must be fully supported (and should not require data block copying) but the metadata subsystem need not be optimized for such operations.

CalvinFS stores file contents in a non-transactional distributed block store analogous to the collection of chunk servers that make up a GFS deployment. We use our extension of Calvin described above to store all metadata, track directory namespaces, and map logical files to the blocks that store their contents.

Both the block store and the metadata store are replicated across multiple datacenters. In our evaluation, we used three physically separate (and geographically distant) datacenters, so our discussion below assumes this type of deployment and refers to each full system replica as being in its own datacenter. However, the replication mechanisms discussed here can just as easily be used in deployments within a single physical datacenter by dividing it into multiple *logical* datacenters.

As with GFS and HDFS, clients access a CalvinFS deployment not via kernel mounting, but via a provided client library, which provides standard file access APIs and file utils [10, 21]. No technical shortcoming prevents CalvinFS from being fully mountable, but implementation of this functionality remains future work.

4 The CalvinFS Block Store

Although the main focus of our design is metadata management, certain aspects of CalvinFS's block store affect metadata entry format and therefore warrant discussion. Most of these decisions were made to simplify the tasks of implementing, benchmarking, and describing the system; other designs of scalable block stores would also work with CalvinFS's metadata architecture.

4.1 Variable-Size Immutable Blocks

As in many other file systems, the contents of a CalvinFS file are stored in a sequence of zero or more blocks. Unlike most others, however, CalvinFS does *not* set a fixed block size—blocks may be anywhere from 1 byte to 10 megabytes. A 1-GB file may therefore legally consist of anywhere from one hundred to one billion blocks, although steps are taken to avoid the latter case.

Furthermore, blocks are completely immutable once written. When appending data to a file, CalvinFS does not append to the file's final block—rather, a new block containing the appended data (but not the original data) is written to the block store, and the new block's ID and size are added to the metadata entry for the file.

[2]Note that this still involves many distributed transactions. For example, creating or deleting a file also updates its parent directory.

4.2 Block Storage and Placement

Each block is assigned a globally unique ID, and is assigned to a block "bucket" by hashing its ID. Each bucket is then assigned to a certain number of block servers (analogous to GFS Chunkservers [10]) at each datacenter, depending on the desired replication factor for the system. Each block server stores its blocks in files on its local file system.

The mapping of buckets to block servers is maintained in a global Paxos-replicated configuration file and changes only when needed due to hardware failures, load balancing, adding new machines to the cluster, and other global configuration changes. Every CalvinFS node also caches a copy of the bucket map. This allows any machine to quickly locate a particular block by hashing its GUID to find the bucket, then checking the bucket map to find what block servers store that bucket. In the event where a configuration change causes this cached table to return stale data, the node will fail to find the bucket at the specified server, query the configuration manager to update its cached table, then retry.

In the event of a machine failure, each bucket assigned to the failed machine is reassigned to a new machine, which copies its blocks from a non-failed server that also stored the reassigned bucket.

To avoid excessive fragmenting, a background process periodically scans the metadata store and compacts files that consist of many small blocks. Once a compacted file is asynchronously re-written to the block store using larger blocks, the metadata is updated—as long as the file contents haven't changed since this compaction process began. If that part of the file *has* changed, the newly written block is discarded and the compaction process restarts for the file.

5 CalvinFS Metadata Management

The CalvinFS metadata manager logically contains an entry for every version (current and historical) of every file and directory to appear in the CalvinFS deployment. The metadata store is structured as key- value store, where each entry's key is the absolute path of the file or directory that it represents, and its value contains the following:

- **Entry type.** Specifies whether the entry represents a file or a directory[3].
- **Permissions.** CalvinFS uses a mechanism to support POSIX hierarchical access control that avoids full file system tree traversal when checking permissions for

[3]Although we see no major technical barrier to supporting linking in CalvinFS, adding support for soft and hard links remains future work.

an individual file by additionally storing all ancestor directories' permissions (up through the / directory) values in tree- ascending order in each metadata entry.

- **Contents.** For directories, this is a list of files and sub-directories immediately contained by the directory. For files, this is a mapping of byte ranges in the (logical) file to byte ranges within specific (physical) blocks in the block store. For example, if a file's contents are represented by the first 100 bytes of block X followed by the 28 bytes starting at byte offset 100 of block Y, then the contents would be represented as [(X, 0, 100), (Y, 100, 28)][4].

To illustrate this structure, consider a directory `fs` in user `calvin`'s home directory, which contains the source files for, say, an academic paper. The `calvinfs-ls` util (analogous to `ls -lA`) yields the following output:

```
$ calvinfs-ls /home/calvin/fs/
drwxr-xr-x calvin users  ...  figures
-rw-r--r-- calvin users  ...  ref.bib
-rw-r--r-- calvin users  ...  paper.tex
```

The CalvinFS metadata entry for this directory would be:

```
KEY:
  /home/calvin/fs
VALUE:
  type:         directory
  permissions:  rwxr-xr-x calvin users
  ancestor-
   permissions: rwxr-xr-x calvin users
                rwxr-xr-x root root
                rwxr-xr-x root root
  contents:     figures ref.bib paper.tex
```

We see that the entry contains permissions for the directory, plus permissions for three ancestor directories: `/home/calvin`, `/home`, and `/`, respectively. Since the path (in the entry's key) implicitly identifies these directories, they need not explicitly named in the value part of the field.

Since permissions checks need not access ancestor directory entries and the `contents` field names all files and subdirectories contained in the directory, the `calvinfs-ls` invocation above only needed to read that one metadata entry. Note that unlike POSIX-style `ls -lA`, however, the command above did not show the sizes of each file. To output those, additional metadata entries have to be read. For example, the metadata entry for `paper.tex` looks like this:

[4]This particular example might come about by the file being created containing 128 bytes in block Y, then having the first 100 bytes overwritten with the contents of block X.

```
KEY:
  /home/calvin/fs/paper.tex
VALUE:
  type:           file
  permissions:    rw-r--r-- calvin users
  ancestor-
    permissions: rwxr-xr-x calvin users
                  rwxr-xr-x calvin users
                  rwxr-xr-x root root
                  rwxr-xr-x root root
  contents:       0x3A28213A 0 65536
                  0x6339392C 0 65536
                  0x7363682E 0 34061
```

Since `paper.tex` is a file rather than a directory, its `contents` field contains block ids and byte offset ranges in those blocks. We see here that `paper.tex` is about 161 KB in total size, and its contents are a concatenation of byte ranges $[0,65536)$, $[0,65536)$, and $[0,34061)$ in three specified blocks in the block store.

Storing all ancestor directories' permissions in each metadata entry eliminates the need for distributed permissions checks when accessing individual files, but comes with a tradeoff: when modifying permissions for a nonempty directory, the new permission information has to be atomically propagated recursively to all descendents of the modified directory. We discuss our protocol for handling such large recursive operations in Section 5.2.

5.1 Metadata Storage Layer

As mentioned above, the metadata management system in CalvinFS is an instance of Calvin with a custom storage layer implementation that includes compound transactions as well as primitive read/write operations. It implements six transaction types:

- `Read(path)` returns the metadata entry for specified file or directory.
- `Create{File,Dir}(path)` creates a new empty file or directory. This updates the parent directory's entry and inserts a new entry for the created file.
- `Resize(path, size)` a file. If a file *grows* as a result of a resize operation, all bytes past the previous file length are by default set to `0`.
- `Write(path, file_offset, source, source_offset, num_bytes)` writes a specified number of bytes to a file, starting at a specified offset within the file. The source data written must be a subsequence of the contents of a block in the block store.
- `Delete(path)` removes a file (or an empty directory). As with the file creation operation, the parent directory's entry is again modified, and the file's entry is removed.

- `Edit permissions(path, permissions)` of a file or directory, which may include changing the owner and/or group.

Each of these operation types also takes as part of its input the user and group IDs of the caller, and performs the appropriate POSIX-style permissions checking before applying any changes. Any POSIX-style file system interaction can be emulated by composing of multiple of these six built-in operations together in a single Calvin transaction.

Three of these six operations (read, resize, write) access only a single metadata entry. Creating or deleting a file or directory, however, touches two metadata entries: the newly created file/directory and its parent directory. Changing permissions of a directory may involve many entries, since all descendants must be updated, as explained above. Since entries are hash-partitioned across many metadata stores on different machines, the create, delete, and change permissions (of a non-empty directory) operations necessarily constitute distributed transactions.

Other operations, such as appending to, copying, and renaming files are constructed by bundling together multiple built-in operations to be executed atomically.

5.2 Recursive Operations on Directories

Recursive metadata operations (e.g., copying a directory, changing directory permissions) in CalvinFS use Calvin's built-in OLLP mechanism. The metadata store first runs the transaction at no isolation in `Analyze` mode to discover the read set without actually applying any mutations. This determines the entire collection of metadata entries that will be affected by the recursive operation by traversing the directory tree starting at the "root" of the operation—the metadata entry for the directory that was passed as the argument to the procedure.

Once the full read/write set is determined, it is added as an annotation to the transaction request, which is repeated in `Run` mode, during which the directory tree is retraversed from the operation to check that the read/write set of the operation has not grown (e.g., due to a newly inserted file in a subtree). If the read- and write-sets have grown between the `Analyze` and `Run` steps, OLLP (deterministically) aborts the transaction, and restarts it again in `Run` mode with an appropriately updated annotation.

6 The Life of an Update

To illustrate how CalvinFS's various components work together in a scalable, fault-tolerant manner, we present the end-to-end process of executing a simple operation—creating a new file and writing a string to it:

```
echo "import antigravity" >/home/calvin/fly.py
```

The first step is for the client to submit the request to a CalvinFS front-end—a process that runs on every CalvinFS server and orchestrates the actual execution of client requests, then returns the results to the client.

Write File Data

After receiving the client request, the front-end begins by performing the write by inserting a data block into CalvinFS's block store containing the data that will be written to the file. The first step here is to obtain a new, globally unique 64-bit block id β from a block store server. β is hashed to identify the bucket that the block will belong to, and the front-end then looks up in its cached configuration file the set of block servers that store that bucket, and sends a block creation request interface node now sends a block write request ($\beta \rightarrow$ `import antigravity`) to each of those block servers.

Once a quorum of the participating block servers (2 out of 3 in this case) have acknowledged to the front- end that they have created and stored the block, the next step is to update the metadata to reflect the newly created file.

Construct Metadata Operation

Since our system does not provide a single built-in operation that both creates a file and writes to it, this operation is actually a compound request specifying three mutations that should be bundled together:

- create file `/home/calvin/fly.py`
- resize the file to 18 bytes.
- write $\beta : [0,18)$ to byte range $[0,18)$ of the file

Once this compound transaction request (let's call it α) is constructed, the front-end is ready to submit it to be applied to the metadata store.

Append Transaction Request to Log

The first step in applying metadata mutation is for the CalvinFS front-end to append α to the log. The CalvinFS front-end sends the request to a Calvin *log* front-end, which appends α to its current batch of log entries, which has some globally unique id γ. When batch γ fills up with requests (or after a specified duration), it is written out another asynchronously replicated block store. Again, the log front-end waits for a majority of block servers to acknowledge its durability, and then does two things: (a) it submits the batch id γ to be appended to the Paxos- replicated metalog, and (b) it goes through the batch in order, forwarding each transaction request to all metadata shards that will participate in its execution.

Apply Update to Metadata Store
Each Calvin metadata shard is constantly receiving transaction requests from various Calvin log front- ends—however it receives them in a completely unspecified order. Therefore, it also reads new metalog entries as they are successfully appended, and uses these to sort the transaction requests coming in from all of the log front-ends, forming the precise subsequence of the log containing exactly those transactions in whose execution the shard will participate. Now, the sequencer at each metadata storage shard can process requests in the correct order.

Our example update α reads and modifies two metadata records: `/home/calvin` and `/home/calvin/fly.py`. Suppose that these are stored on shards P and Q, respectively. Note that each metadata shard is itself replicated multiple times—once in each datacenter in the deployment—but since no further communication is required between replicas to execute α, let us focus on the instantiations of P and Q in a single datacenter (P_0 and Q_0 in datacenter 0, say).

Both P_0 and Q_0 receive request α in its entirety and proceed to perform their parts of it. At P_0, α requests a lock on record `/home/calvin` from the local scheduler; at Q_0, α requests a lock on `/home/calvin/fly.py`. At each machine, α only starts executing once it has received its local locks.

Before we walk through the execution of α at P_0 and Q_0, let us first review the sequence of logical steps that the request needs to complete:

1. **Check parent directory permissions.** Abort transaction if `/home/calvin` does not exist or is not writable.
2. **Update parent directory metadata.** If `fly.py` is not contained in `/home/calvin`'s contents, add it.
3. **Check file permissions.** If the file exists and is not a writable file, abort the transaction.
4. **Create file metadata entry.** If no metadata entry exists for `/home/calvin/fly.py`, create one.
5. **Resize file metadata entry.** Update the metadata entry to indicate a length of 18 bytes. If it was previously longer than 18 bytes, this truncates it. If it was previously shorter (or empty), it is extended to 18 bytes, padded with zeros.
6. **Update file metadata entry's contents.** Write $\beta : [0,18)$ to the byte range $[0,18)$ of `/home/calvin/fly.py`, overwriting any previously existing contents in that range.

Note that steps 1 and 2 involve the parent directory metadata entry at P_0, while steps 3, 4, 5, and 6 involve only the new file's metadata at Q_0. However, steps 4 through 6 depend on the outcome of step 1 (as well as 3), so P_0 and Q_0 do need to coordinate in their handling of this mutation request. The two shards therefore proceed as follows:

P_0	Q_0
Check parent dir permissions (step 1).	Check file permissions; abort if not OK (step 3).
Send ***result*** (OK or ABORT) to Q_0.	Receive parent directory permissions check ***result*** from P_0.
If ***result*** was OK, update parent dir metadata (step 2).	If received ***result*** is OK, perform steps 4 through 6.

Both shards begin with permissions checks (step 1 for P_0 and step 3 for β). Suppose that both checks succeed. Now α sends an OK ***result*** message to β. β receives the ***result*** message, and now both shards execute the remainder of the operation with no further coordination.

Note that we were discussing datacenter 0's P and Q metadata shards. Metadata shards (P_1,Q_1), (P_2,Q_2), etc., in other datacenters independently follow these same steps. Since each shard deterministically processes the same request sequence from the log, metadata state remains strongly consistent: `import antigravity` is written to `/home/calvin/fly.py` identically at every datacenter.

7 Performance Evaluation

CalvinFS is designed to address the challenges of (a) distributing metadata management across multiple machines, and (b) wide area replication for fault tolerance. In exploring the scalability and performance characteristics of CalvinFS, we therefore chose experiments that explicitly stressed the metadata subsystem to its limits.

WAN replication. All results shown here used deployments that replicated all data and metadata three ways—across datacenters in Oregon, Virginia, and Ireland.

Many small data blocks. In order to test the performance of CalvinFS's metadata store (as opposed to the more easily scalable block storage component), we focused mainly on update-heavy workloads in which 99.9% of files were 1KB or smaller. Obviously, most real world file systems typically deal with much larger files; however, by experimenting on smaller files we were able to test the ability of the metadata store to handle billions of files while keeping the cluster size affordably small enough for our experimental budget. Obviously, larger files would require additional horizontal scalability of the block store; however this is not the focus of our work.

We use this setup to examine CalvinFS's memory usage, throughput capacity, latency, and fault tolerance.

7.1 Experimental Setup

All experiments were run on EC2 High-CPU Extra-Large instances[5]. Each deployment was split equally between AWS's US-West (Oregon), US-East (Virginia), and EU (Ireland) regions. Block and metadata replication factors were set to 3, and buckets, metadata shards, and Paxos group members were placed such that each object (metadata entries, log blocks, data blocks, and Paxos log and metalog entries) would be stored once in each datacenter.

Each machine served as (a) a block server (containing 30 buckets), (b) a log front-end, and (c) a metadata shard. In addition, one randomly selected machine from each datacenter participated in the Paxos group for the Calvin metalog. We ran our client load generation program on the same machines (but it did not use any knowledge about data or metadata placement when generating requests, so very few requests could be satisfied locally, especially in large deployments).

We ran each performance measurement on deployments of seven different sizes: 3, 6, 18, 36, 75, 150, and 300 total machines. As mentioned above, we had a limited budget for running experiments, so we could not exceed 300 machines. However, we were able to store billions of files across these 300 machines by limiting the file size. Our results can be translated directly to larger clusters that have more machines and larger files (and therefore the same total number of files to manage). We compare our findings directly to HDFS performance measurements published by Yahoo researchers [20].

7.2 File Counts and Memory Usage

After creating each CalvinFS deployment, we created 10 million files per machine. File sizes ranged from 10 bytes to 1MB, with an average size of 1kB. 90% of files contained only one block, and 99.9% of files had a total size of under 1kB. Most file names (including full directory paths) were between 25 and 50 bytes long.

We found that total memory usage for metadata was approximately 140 bytes per metadata entry—which is closely comparable to the per-file metadata overhead of HDFS [28]. Unlike HDFS, however, the metadata shards did *not* store an in-memory table of block placement data, since Calvin uses a coarser-grained bucket placement mechanism instead. We would therefore expect an HDFS-like file system deployment (with ˜1 block per file) to require approximately twice the amount of total memory to store metadata (assuming the same level of metadata replication). Of course, by partitioning metadata across machines, CalvinFS requires far less memory **per machine**.

[5]Each EC2 High-CPU Extra-Large instance contains 7 GB of memory, 20 EC2 Compute Units (8 virtual cores with 2.5 EC2 Compute Units each with the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor), and 1690 GB of instance storage

Our largest deployment—300 machines—held 3 billion files (and therefore 9 billion total metadata entries) in a total of 1.3 TB of main memory. This large number of files is far beyond what HDFS can handle [27].

7.3 Throughput Capacity

Next, we examined the throughput capacity (Figure 1) and latency distributions (Figure 2) of reading files, writing to files, and creating files in CalvinFS deployments of varying sizes. For each measurement, we created client applications that issued requests to read files, create files, and write to files—but with different frequencies. For experiments on read throughput, 98% of all client requests were reads, with 1% of operations being file creations and 1% being writes to existing files. Similarly, for write benchmarks, clients submitted 98% write requests, and for append benchmarks, clients submitted 98% append requests. For all workloads, clients chose which files to read, write, and create using a Gaussian distribution.

Once key feature of CalvinFS is that throughput is totally unaffected by WAN replication (and the latencies of message passing between datacenters). This is because once a transaction is replicated to all datacenters by the Calvin log component (which happens before request execution begins), no further cross-datacenter communication is required to execute the transaction to completion. Therefore, we only experiment with the three datacenter case of Oregon, Virginia, and Ireland for these set of experiments—changing datacenter locations (or even removing WAN replication entirely) has no effect on throughput results. Latency, however, is affected by the metalog Paxos agreement protocol across datacenters, which we discuss in Section 7.4 below.

Read Throughput

For many analytical applications, extremely high read *throughput* is extremely important, even if it comes at the cost of occasionally poor latencies for reads of specific files. On the other hand, being able to rely on consistent read latencies vastly simplifies the development of distributed applications that face end-users. We therefore performed two separate read throughput experiments: one in which we fully saturated the system with read requests, resulting in "flaky" latency, and one at only partial load that yields reliable (99.9th percentile) latency (Figures 1a and 1b). Because each datacenter stores a full, consistent replica of all data and metadata, each read

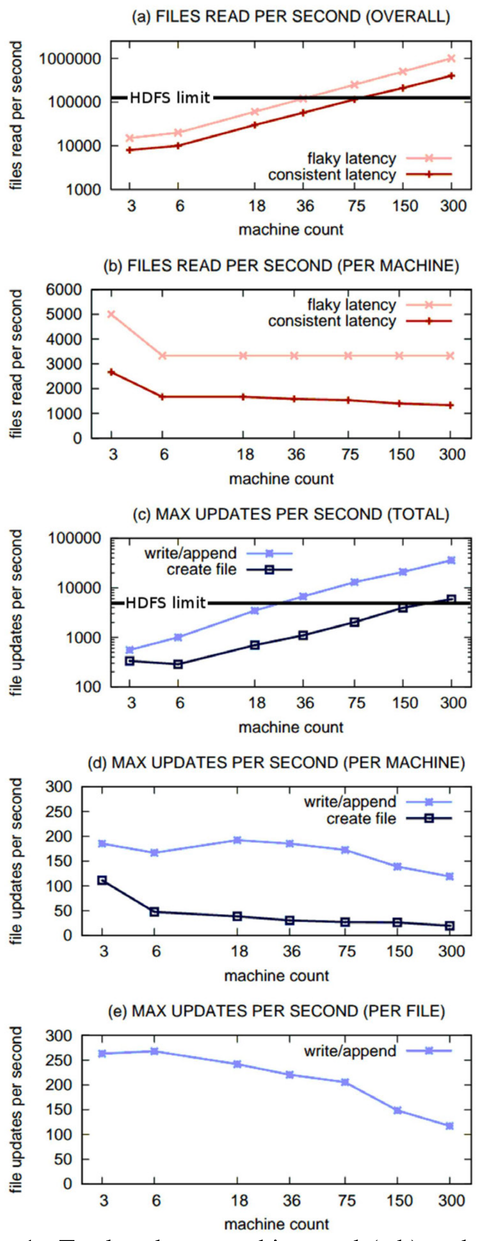


Figure 1: Total and per-machine read (a,b) and update (c,d) throughput, and maximum per-file update throughput (e), for WAN-replicated CalvinFS deployments.

request is routed only to the relevant machine(s) in the same datacenter as the client.

Specifically, we observed that under very heavy load, occasional background tasks such as LSM tree compactions and garbage collection could cause a large number of concurrent read requests to stall, introducing occasional latency spikes and some completely failed reads that then had to be retried. Median and 90th percentile latencies, however, were comparable to those observed for the "partial load" experiments described below.

For our partial load experiments, we reduced the number of clients as far as necessary to completely remove latency spikes. Typically running the system at 50% of the maximum load accomplished this. For our largest deployments, we had to reduce the load to about 45% of maximum throughput to accomplish this [6].

Figures 1a and 1b) show that CalvinFS is able to achieve linear scalability for read throughput, even as millions of files are read per second. At machine count 3, there is only one machine per datacenter, so all reads can be satisfied locally, which yields very high throughput. Starting with machine count 6, however, the probability of at least one non-local access increases rapidly (already at machine count 6 there is a 75% probability that either the file metadata or the file data itself will be non-local).

We include in Figure 1a the upper bound of read request throughput for HDFS, as reported by Yahoo researchers in 2010[20]. Specifically, this corresponds to block location lookups by the NameNode. It was found that the HDFS metadata store can serve no more than 126,119 block location lookups per second. Since read requests involve more metadata operations than just a single block location lookup—such as other metadata entry lookups to check file existence, permissions, and block IDs, not to mention possibly having to look up multiple block locations if the file spans multiple blocks—this is strictly an upper bound. We also assume here that the metadata management layer is the *only* bottleneck for reads, which in HDFS would certainly not be the case for small deployments. It is fair to expect actual HDFS read throughput to be considerably lower than the upper bound plotted in Figure 1a.

Update Throughput

Next, we measured the total number of file creation and append operations that each CalvinFS deployment could perform (Figures 1c and 1d). Append throughput scaled very nearly linearly with the number of machines in the cluster, reaching about 40,000 appends per second with a 300- machine cluster.

However, file creation throughput scaled slightly less smoothly. This is because each file creation operation is implemented as a distributed transaction (since metadata entries had to be modified for both the parent directory and the newly-created file)—requiring coordination between metadata shards to complete. As more machines

[6]The problems of performance isolation between processes and mitigating tail latencies have been studied extensively, and many techniques have been developed that could be applied to CalvinFS to safely increase CPU utilization and improve performance, but these are outside the scope of this paper.

are added, the likelihood increases that at any given time at least one machine will "fall behind," delaying other machines' progress involved in distributed transactions accessing data at that node. This effect is similar to that observed running TPC-C and other OLTP benchmarks using the Calvin framework [24]. Despite slightly sub-linear scaling, file creation capacity in CalvinFS still scales very well overall—far better than alternatives that do not scale metadata management across multiple machines.

We include in Figure 1c the observed HDFS throughput upper bound of 5600 new blocks per second [20]. In our benchmark, each file creation and write involved creating a block, then performing one or more other metadata updates. We therefore expect actual HDFS update throughput of this type to be considerably lower than 5600 operations per second, but this figure serves as a proven upper bound.

Concurrent Writes to Contended Files

Many distributed systems do not perform well when a large number of clients concurrently attempt to write to the same file—such as heavy traffic of simultaneous appends to a shared log file. The systems that provide the best performance for this situation often forgo consistent replication and strong linearizability guarantees to do so.

CalvinFS, however, supports high concurrent write throughput to individual files without sacrificing linearizability. To demonstrate this, we performed an experiment in which we chose one file for every three machines in the full deployment (so 1 file for the 3-machine deployment and 100 files for the 300 machine deployment) and had 100 independent clients per file repeatedly send requests to either append data to that file or perform a random write within the file. Figure 1e shows the resulting per-file throughput. Small-cluster CalvinFS deployments sustained rates of 250 writes or appends per second on each file. Our largest deployments sustained 130 writes or appends per second on each file.

7.4 Latency Measurements

Next, we examined the latency distribution for file read, write, and file creation operations for deployments of 36 and 300 machines (Figures 2a and 2b). Latencies are measured from when a client submits a request until the operation is completed and it receives a final response.

Read Latencies

Our measurement of read latencies was taken under "non-flaky" load, which is about half of maximum read throughput. In all cases, read requests are served by the nearest metadata shard and block server within the same datacenter.

We broke reads down into three categories: (a) reads of files that contain no data in the block store (this includes `ls` operations on directories, since each directory's contents are listed in its metadata entry), reads of files that contain a single block, and reads of multi-block files.

There are several interesting features to note in these plots. First, in the 36-machine deployment, the median latency to read a non-empty file is about 3ms and the 99th percentile latency is about 80ms, while at 300 machines, median latency is about 5 ms, and 99th percentile latency is about 120ms. Although adding more machines to a distributed system invariably introduces performance variabilities, we deemed this a reasonable latency price for nearly an order of magnitude of scaling.

Second, when reading zero-block files in the 36-machine deployment (which has 12 machine per datacenter), about 1 read in 12 is extremely fast—less than 100 microseconds—because 1 in 12 metadata lookups happen to occur on the same machine as the interface node handling the client's request, requiring no network round trips. The same effect is visible for one 100th of metadata-only reads in the 300-machine deployment (which, likewise, has 100 machines per datacenter). LAN round-trip times within a datacenter were 1 ms—about the latency of most non-local metadata-only reads. Similarly, 1-block reads generally incur 2 round trips, while 2+ block reads incur 3 or more. Because of the non-uniform distribution of files read, around 85% of blocks could be served directly from block servers' OS memory cache, without needing to go to disk; only 15% of 1-block reads incur disk I/O costs; among reads of multi-block files, the frequency of I/O latencies appearing is higher.

Although these benchmarks may not be very indicative of real-world usage patterns for distributed file systems (which would likely include many more large files, and in some cases worse cache locality for reads), we chose them to highlight the specific sources of latency that are introduced by components *other* than the block store. Therefore, one can know what to expect if a CalvinFS-style metadata subsystem were coupled with an off-the-shelf block store whose performance and scalability was already well-documented for petabyte-scale data volumes and much larger individual block sizes.

Update Latencies

Latencies for file creation and write/append requests are dominated by WAN round-trip times. Creating a file typically incurs approximately two non-overlapping round trip latencies: one for the log front-end to write its request batch out to a majority of datacenters, and one to append the entry to the Paxos metalog.

Although we saw above that CalvinFS achieves more

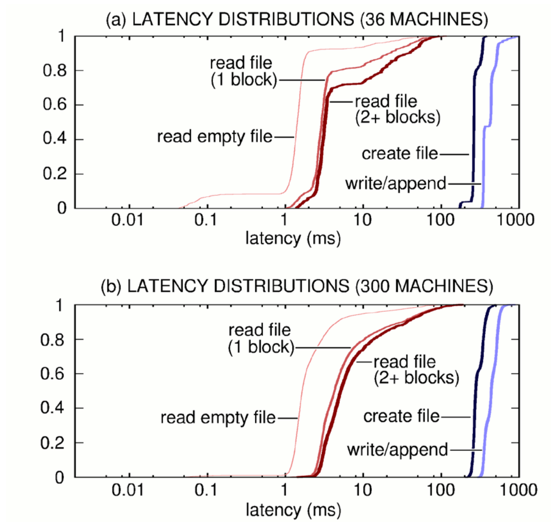


Figure 2: Latency distributions for read, write/append, and create operations for WAN-replicated CalvinFS deployments of (a) 36 machines and (b) 300 machines.

impressive *throughput* for writes and appends to existing files than for file creation, the latency for writes/appends is higher—one additional non- overlapping WAN round trip is necessary to replicate the newly created data blocks before requesting the metadata update.

7.5 Fault Tolerance

Since we designed CalvinFS's WAN replication mechanism with the explicit goal of high availability, we now test our system in the presence of full datacenter failure. In our next experiment, we killed all CalvinFS processes in the Virginia datacenter while a 36-machine CalvinFS deployment the system was running under a mixed read/create/write load. Specifically, we deployed 1000 clients—one third constantly reading files, one third constantly creating new files, and one third constantly appending to files. This saturates the file system's maximum file creation throughput capacity (which is limited by lock contention) and represents approximately 50% read load and 20% append load.

Figure 3 shows throughput (a) and median and 99th-percentile latency (b) for the 30 seconds immediately preceding and following the datacenter "failure". In order to clearly show what effects this had on CalvinFS's core operation capacities, we *immediately* redirected all new client requests that would have been routed to Virginia to either Oregon or Ireland, rather than requiring clients to wait for timeouts before resuming (which would have "unfairly" given the system time to recover from its sudden involuntary reconfiguration).

We see here that total read, create, and write/append throughput capacity is only reduced by a small amount,

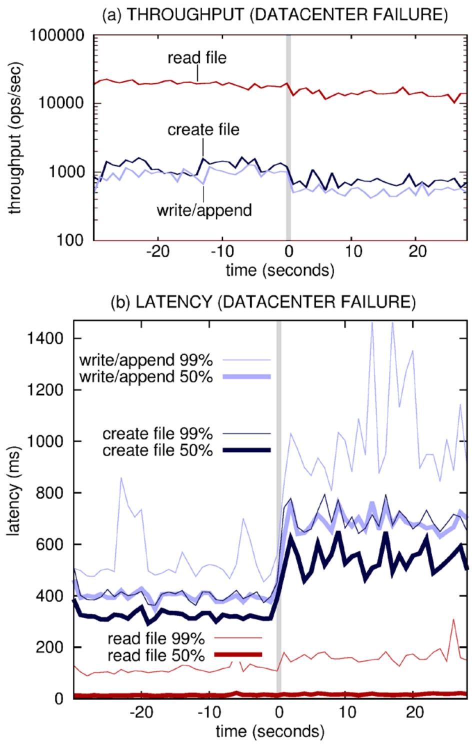


Figure 3: Throughput (a) and latency (b) for the time window preceding and following a datacenter failure.

median read latency remains unchanged, and 99th-percentile read latency only increases by about 30%. File creation and write/append latency, however, roughly double. The reason for this is that the non-overlapping portions of WAN latencies goes from being around 100ms (round trip between either Oregon and Virginia or Virginia and Ireland—each of which pair forms a quorum) to nearly 200ms (round trip between Oregon and Ireland, which now represent the only quorum). No file is at any time unavailable for reading or writing.

In summary, we found that CalvinFS tolerated an unplanned datacenter outage with exceptional grace.

8 Related Work

CalvinFS builds on a long history of research on the scalability and reliability of distributed file systems.

We modeled certain aspects of the CalvinFS design after GFS/HDFS. In particular, our decision to concentrate all metadata in the main memory of a specific metadata component is based on the success of this tactic in GFS/HDFS. CalvinFS' block store is also a simplification of the GFS/HDFS model that uses consistent hashing to simplify block metadata. Our implementation of CalvinFS' novel features—scalable metadata management and consistent WAN replication—was designed to illuminate a path that a GFS-/HDFS-like file system could take towards eliminating the single metadata master as both a scalability bottleneck and availability hazard [10, 21, 20].

In 2009, Google released a retrospective on the scalability, availability, and consistency challenges that GFS had faced since its creation, attributing many difficulties to its single-master design. The interview also describes a new distributed-master implementation of GFS that stores metadata in Bigtable [10, 14, 6]. Since Bigtable supports neither multi-row transactions nor synchronous replication, it is unclear how (or if) this new GFS implementation supports strongly consistent semantics and linearizable file updates while maintaining high availability—particularly in the case of machine failures in the metadata Bigtable deployment.

The Lustre file system resembles GFS in that it uses a single metadata server (MDS), but it does not store per-block metadata, reducing MDS dependence in the block creation and block-level read paths. The latest release of Lustre allows metadata for specific directory subtrees to be offloaded to special "secondary" MDSs for outward scalability and load balancing. Lustre supports only cluster-level data replication [18].

Tango provides an abstraction of a distributed, transactional data structure backed by a replicated, flash-resident log, and is designed for use in metadata subsystems. Like in the CalvinFS metadata manager, a Tango deployment's state is uniquely determined by a single serialized log of operation requests. Tango transactions use optimistic concurrency control, however: they log a commit entry as the final execution step (readers of the log are instructed to ignore any commit entry that turns out to be preceded by a conflicting one). To avoid high optimistic abort rates under contention, this mechanism requires a log implementation with very low append latency. Since synchronous geo-replication inherently incurs high latencies, Tango is only suited to single-datacenter deployments [5].

IBM's GPFS distributes metadata using a shared-disk abstraction and allows multiple machines to access it concurrently protected by a distributed locking mechanism. When multiple clients access the same object, however, distributed locking mechanisms perform poorly. File systems that store metadata using shared-disk arrays depend on low-latency network fabrics to mitigate these issues [17].

Gluster distributes and replicates both data blocks and file metadata entries using an elastic hashing algorithm. However, adding replicas to a Gluster deployment significantly hinders write throughput. Furthermore, Gluster's implementation of copy and rename operations forces data blocks as well as metadata to be copied between storage shards, which can easily become too expensive [1].

The Ceph file system scales metadata management by dynamically partitioning metadata by directory subtree (and hashing "hotspot" directories across multiple metadata servers). Ceph is optimized for single- datacenter deployments, however; its metadata replication mechanism relies heavily on low latencies between replicas to avoid introducing update-contention bottlenecks [25].

The Panasas File System colocates file metadata with file data on Object-based Storage Devices (OSDs), each of which manages (RAID-based) data replication independently. OSDs optimize caching for high-throughput concurrent reads. Clients cache a global mapping of file system objects to OSDs, updates to which require global synchronization [26].

The Ursa Minor Storage System uses subtree-partitioning to distribute metadata but takes a different approach to outwardly scalable metadata management: any time an atomic operation would span multiple partitions, instead of using a distributed transaction, it repartitions the metadata data, migrating all entries that need to be atomically updated to the same partition [3].

The Farsite file system is designed to unite a collection of "desktop" computers rather than datacenters full of rack servers. Early versions Farsite relied on a single metadata server, but Farsite now supports dynamic subtree-partitioning as well, but no metadata replication [4].

Frangipani and xFS are shared-disk distributed file systems. xFS implemented a "serverless" file system, distributing file data and metadata across a collection of disks, using a globally replicated mapping of file system object locations. All implementation logic is executed by clients, using on-disk state for synchronization. Some currently popular shared-disk-based file systems appear to be loosely based on the xFS design [23, 22].

Panache approaches file system scalability from a different direction—providing scalable caching of both data and metadata for a traditional (and less scalable) file system. Although Panache does not provide a full replacement for a file system's metadata component, it effectively removes some bottlenecks, particularly from the read path, via partitioning and replication [9].

Like CalvinFS, Giga+ uses hash partitioning to distribute metadata for across many servers within a datacenter. However Giga+'s distributed operations are eventually consistent and rely on clever handling of stale client-side state [16].

9 Conclusions

CalvinFS deployments can scale on large clusters of commodity machines to store billions of files and process hundreds of thousands of updates and millions of reads per second—while maintaining consistently low read latencies. Furthermore, CalvinFS deployments can survive entire datacenter outages with only minor performance consequences and no loss of availability at all.

References

[1] Glusterfs. *Gluster Community, http://gluster.org/.*

[2] Riak. *Basho, http://basho.com/riak.*

[3] M. Abd-El-Malek, W. V. Courtright II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. P. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, et al. Ursa minor: Versatile cluster-based storage. In *FAST*, volume 5, 2005.

[4] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. *ACM SIGOPS Operating Systems Review*, 36(SI):1–14, 2002.

[5] 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 *SOSP*, 2013.

[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. *TOCS*, 2008.

[7] 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*, pages 251–264, 2012.

[8] 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. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, Oct. 2007.

[9] M. Eshel, R. L. Haskin, D. Hildebrand, M. Naik, F. B. Schmuck, and R. Tewari. Panache: A parallel file system cache for global file access. In *FAST*, 2010.

[10] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proc. of SOSP*, 2003.

[11] F. Hupfeld, T. Cortes, B. Kolbeck, J. Stender, E. Focht, M. Hess, J. Malo, J. Marti, and E. Cesario. The xtreemfs architecture: a case for object-based file systems in grids. *Concurrency and computation*, 20(17), 2008.

[12] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 2010.

[13] P. I. LLC. National survey on data center outages. 2010.

[14] M. K. McKusick and S. Quinlan. Gfs: Evolution on fast-forward.

[15] R. McMillan. Guns, squirrels, and steel: The many ways to kill a data center. *Wired*, 2012.

[16] S. Patil and G. A. Gibson. Scale and concurrency of giga+: File system directories with millions of files. In *FAST*, 2011.

[17] F. B. Schmuck and R. L. Haskin. Gpfs: A shared-disk file system for large computing clusters. In *FAST*, volume 2, 2002.

[18] P. Schwan. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux Symposium*, 2003.

[19] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. O. K. Littlefield, D. Menestrina, S. E. J. Cieslewicz, I. Rae, et al. F1: A distributed sql database that scales. *Proceedings of the VLDB Endowment*, 6(11), 2013.

[20] K. Shvachko. Hdfs scalability: the limits to growth. 2009.

[21] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *MSST*, 2010.

[22] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the xfs file system. In *USENIX Annual Technical Conference*, volume 15, 1996.

[23] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: a scalable distributed file system. *SIGOPS Oper. Syst. Rev.*, 1997.

[24] A. Thomson, T. Diamond, S. chun Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *SIGMOD*, 2012.

[25] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: a scalable, high-performance distributed file system. In *OSDI*, 2006.

[26] B. Welch, M. Unangst, Z. Abbasi, G. A. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable performance of the panasas parallel file system. In *FAST*, volume 8, 2008.

[27] T. White. The small files problem. *Cloudera Blog, blog.cloudera.com/blog/2009/02/the-small-files-problem/.*

[28] T. White. *Hadoop: The Definitive Guide*. 2nd edition, 2010.

Analysis of the ECMWF Storage Landscape

Matthias Grawinkel, Lars Nagel, Markus Mäsker, Federico Padua, André Brinkmann
Johannes Gutenberg University Mainz

Lennart Sorth
ECMWF

Abstract

Despite domain-specific digital archives are growing in number and size, there is a lack of studies describing their architectures and runtime characteristics. This paper investigates the storage landscape of the European Centre for Medium-Range Weather Forecasts (ECMWF) whose storage capacity has reached 100 PB and experiences an annual growth rate of about 45%. Out of this storage, we examine a 14.8 PB user archive and a 37.9 PB object database for metereological data over a period of 29 and 50 months, respectively.

We analzye the system's log files to characterize traffic and user behavior, metadata snapshots to identify the current content of the storage systems, and logs of tape libraries to investigate cartridge movements. We have built a caching simulator to examine the efficiency of disk caches for various cache sizes and algorithms, and we investigate the potential of tape prefetching strategies. While the findings of the user archive resemble previous studies on digital archives, our study of the object database is the first one in the field of large-scale active archives.

1 Introduction

The number and size of computing sites with domain-specific archives has reached new heights and is still increasing. Next to the ever faster compute systems, storage systems are also growing in multiple dimensions like available capacity, access frequency, and required throughput. With the increased computing power and new algorithms from the big data area, computations tend to use and create more data. Here, many archives become active in the sense that every stored datum may be read at any point in time.

For building storage systems that meet the demands of active archives, it is necessary to understand how today's systems evolved, how they work and in which direction the development is heading. Unfortunately, only a small number of publicly available studies exist that analyze the storage infrastructure and characterize the stored data as well as storage access patterns and growth patterns. The result is a lack of representability of previous studies, as comparable studies are missing. Most of today's multi-petabyte storage systems follow a tape backend + disk caching approach. While disks offer the better performance and more flexibility in their access characteristics, tape is still cheaper in terms of capacity. The disk-to-tape ratio is therefore a tradeoff between price, performance, and capacity.

Previous studies investigated traces of desktop or network file systems [19, 28], internet accessible content delivery networks [15, 17], in-memory caches [3], or digital archives and content repositories [21, 1, 14]. The presented study is the first analysis of an active archive – a large-scale content repository where all data is subject to be accessed at any time.

The contributions of this paper are an in-depth system analysis of two archival systems from the previously uncharted weather forecasting domain, a simulator-driven evaluation of workload trace files to improve the disk cache efficiency, and a feasibility evaluation of tape prefetching strategies. The subject of our study is the storage environment of the *European Centre for Medium-Range Weather Forecasts* (ECMWF)[1], which provides medium-range global weather forecasts for up to 15 days and seasonal forecasts for up to 12 months. To achieve this, they utilize supercomputers[2] and, as of September 2014, storage with a capacity of 100 PB. Next to fast HPC storage, they run two in-house developed archival systems: a general-purpose user-accessible archive (ECFS) for file storage hosting 14.8 PB of data and a large object database for meteorological data (MARS) that hosts 37.9 PB of primary data consisting of 170 billion fields. It is regarded as the world's largest

[1]http://www.ecmwf.int/
[2]http://www.top500.org/site/47752

archive of numerical weather prediction data. Both systems consist of multiple tape libraries with disk-based caches in front of them. We have developed a trace-based storage simulator for the ECFS traces to determine the efficiency of various cache strategies and to optimize the hit-ratio of the disk caches. Additionally, we look into the logs of the tape libraries and the backend HPSS system [12]. Examining logs with more than 9.5 million tape load operations in 2012-2013, we investigate the feasibility of tape prefetching strategies.

Our study shows that the two storage systems are used in different ways. While the ECFS is an archive with mostly write accesses and only a small set of actively used data, the MARS system is read-dominant, and all its data are subject to be read. Both systems face an exponential data increase with a compound annual growth rate (CAGR) of about 45% over the last years and about 50% today. In total, the ECFS logs cover 29 months of 2012-2014 and the MARS logs 50 months of 2010-2014. These logs cover the integration of new applications, models and hardware. Especially, the additional throughput and capacity required by a newly commissioned supercomputer becomes visible at several points and is one of the reasons for the exponential growth in storage capacity.

2 Related Work

Large-scale storage and archival systems have been investigated for many years. Baker et al. and Rosenthal et al., for example, discuss the technical and non-technical challenges for building long-term digital repositories [4, 5, 24]. The technical problems include large-scale disasters, component and media faults, and the obsolescence of hardware, software and formats. Furthermore, human errors, loss of data context, or misplanning need to be considered. Rosenthal especially emphasizes the economical aspects of long-term archives. Economic faults, erroneous capacity planning, or the wrong use of storage technologies can be a threat for long-term data availability. Many previous studies help to encounter these threats and to build reliable and successful digital archives.

Most studies that analyze the contents or behavior of file systems deal with workstations, general-purpose network file systems, or HPC storage systems [13, 2, 19, 11, 8, 28]. They examine static file system snapshots, request traces and operating system logs to investigate multiple dimensions of storage systems. Meister et al. investigated the possible impact of applied deduplication on HPC storage [23].

Another investigated area are large-scale publicly accessible systems, their usage patterns, and the efficiency of caching [7, 25, 15, 3, 17]. It is especially important to understand and characterize traffic and user behavior to build and improve caching infrastructures.

There exist only a few publicly available archival traces [20] and analyses of recent archives. Madden et al. wrote a technical report on the user behavior in the NCAR archival system over a three year period from 2008 to 2010 [21]. They also started an investigation of namespace locality of user sessions. Frank et al. compare the logs of an NCAR system to a previous study of the system from 1992 [14]. In the interval, the read-to-write ratio on the system changed from 2:1 to 1:2 which indicates that archives are becoming increasingly write-only. From the traces it was also derived that 30% of the requests have a *latency to first byte* of more than three minutes. In order to improve the latency, the authors suggest large disk caches that permanently hold the small files. The most comprehensive study was conducted by Adams et al. [1] who examined multiple public and scientific long-term data repositories for their content and workload behavior. Especially for the scientific LANL and NCAR repositories, disks play an increasingly important role, which becomes visible by comparing the 1:262 disk-to-tape ratio at NCAR in 1993 with the 1:3.3 ratio at LANL in 2010.

Previous studies document the rise of disk drives, used either as a complement to or as a replacement for tape in large-scale archival scenarios. Colarelli and Grunwald, for example, argue for the replacement of tape archives by large disk arrays that are switched off when not in use [10]. This idea was refined in the Pergamum system by Storer et al., a distributed system of powered-down disks for archival workloads [26]. Grawinkel et al. proposed a high-density MAID system optimized for “write once, read sometimes” workloads [16]. Today, large-scale archival systems are in production that primarily build on disk technology, like the Internet Archive [18].

3 Background

The European Centre for Medium-Range Weather Forecasts (ECMWF) is an independent intergovernmental organization supported by 20 European member states and 14 co-operating states. The center was established in 1975 and hosts one of the largest supercomputer complexes in Europe. Storage for the computation environments is driven by HPC storage systems and two large archival systems developed in-house, namely ECFS and MARS, that will be investigated in this paper. Today the center hosts a combined storage capacity of about 100 PB. This includes the HPC storage and backups. All files of the archival system are stored on tape, and important files are stored to a second tape copy.

ECFS is used as a general-purpose archival system and accessible for users of the ECMWF compute environment. In September 2014 it stored 137 million files with a total size of 14.8 PB. The system provides 0.34 PB of disk caches so that the disk-to-tape ratio is 1:43. All data is written to a disk cache first before it is migrated to tape.

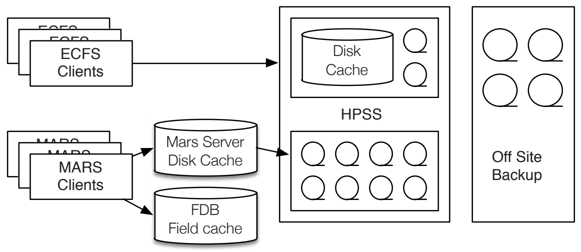


Figure 1: Abstract overview of storage environment.

MARS is an object store for meteorological data with a database-like API. A custom query language is used to specify a list of relevant fields. The system assembles these fields into a package and stores it on a target storage system so that it can be accessed from the HPC systems. To reduce the metadata overhead and keep the file backend manageable on tape, fields are stored in appendable files. A dedicated database allocates and maps fields to offset-length pairs on files. New, recently, and often used fields are staged and cached in a dedicated field database (FDB), which is the primary target for queries. If the FDB does not contain a field, the MARS servers are queried. MARS manages its own disk cache. In case of a cache miss, the required files are loaded from tape. All MARS servers for all projects provide a total of 1 PB of disk cache which results in a 1:38 disk-to-tape ratio. The FDBs are stored on the HPC storage systems connected to the supercomputers and can grow to multiple PB. In September 2014 MARS stored 54 PB of primary data consisting of 170 billion fields in 11 million files. Additionally, the system uses 800 GB of metadata. Each day, 200 million new fields are added.

Figure 1 provides a high level overview of the storage environment, which is implemented around the High Performance Storage System[3] (HPSS) that provides the disk caches for ECFS and manages the tape resources for both ECFS and MARS. Tape is considered to be the final destination for data. Every cached file has a copy on tape. In contrast to ECFS, MARS manages its own disk caches outside of HPSS. The ECMWF runs multiple project-specific MARS databases that are mapped to individual storage pools in the HPSS system. In the following, we will treat ECFS and MARS as two different storage systems.

[3]http://www.hpss-collaboration.org/

3.1 Available Log Files

This work analyzes log files and database snapshots provided by ECMWF. All log files were obfuscated by replacing user information and each part of a file's path by a hash sum, except file extensions. No user information can be revealed, but access patterns and file localities are preserved. We developed a set of python scripts to obfuscate, sanitize, and pack the raw source data. The following analysis is based on compressed log files from multiple ECFS, MARS, and HPSS servers. The gathered and investigated files are:

ECFS access trace: Timestamps, user id, path, size of GET, PUT, DELETE, RENAME requests. 2012/01/02 - 2014/05/21.

ECFS / HPSS database snapshot: Metadata snapshot of ECFS on tape. Owner, size, creation/read/modification date, paths of files. Snapshot of 2014/09/05.

MARS feedback logs: MARS client requests (ARCHIVE, RETRIEVE, DELETE). Timestamps, user, query parameters, execution time, archived or retrieved bytes and fields. 2010/01/01-2014/02/27.

MARS / HPSS database snapshot: Metadata snapshot of MARS files on tape. Owner, size, creation/read/modification date, paths of files. Snapshot of 2014/09/06.

HPSS WHPSS logs / robot mount logs: Timestamps, tape ids, information on full usage lifecycle from access request till cartridges are put back to the library. 2012/01/01 - 2013/12/31.

The traces and tools used are publicly available as outlined in Section 9.

4 ECFS User Archive

Users of the ECMWF compute environment use ECFS as an intermediate and long-term storage for general purpose data. New and recently retrieved files are stored in disk pools and are migrated to the tape storage by HPSS. Files are categorized by their size and are spread to six pools with different capacities and properties. The ranges as well as the number and size of stored files are listed in Table 1. Though tape is considered as the primary storage, files of the *Tiny* class are primarily stored on mirrored disks and only backed up to tape. Therefore, to read tiny files, tape is never used. In ECFS, no files are updated in place, but a file may be overwritten.

4.1 Metadata Snapshot Analysis

In contrast to the trace files that only yield data being accessed within the investigated time frame, the HPSS

Group	From	To (incl.)	Count	Used Capacity
Tiny	0	512 KB	36.0 mil.	4.4 TB
Small	512 KB	1 MB	9.1 mil.	6.3 TB
Medium	1 MB	8 MB	29.5 mil.	101 TB
Large	8 MB	48 MB	30.0 mil.	585 TB
Huge	48 MB	1 GB	29.7 mil.	6.2 PB
Enormous	1 GB	∞	3.1 mil.	8 PB

Table 1: File size categorization. Count and capacity refer to Sept. 2014.

database snapshot gives a full view on all stored files on 2014/09/05.

File system stats	
Total #files	137.5 mil.
Total used capacity	14.8 PB
Largest file size	32 GB
#Directories	5.5 mil.
Max files per directory	0.43 mil.
#Files never read from tape	101.3 mil. (11.3 PB)

Most common file types	
by file count	by used capacity
unknown (27.8232%)	unknown (39.3306%)
.gz (20.4319%)	.tar (21.2699%)
.tar (7.8015%)	.gz (12.4954%)
.nc (7.6312%)	.nc (7.8819%)
.grib2 (1.9438%)	.lfi (2.2399%)
.raw (1.7284%)	.pp (1.0087%)
.txt (1.5095%)	.sfx (0.9327%)
.Z (1.4862%)	.grb (0.8471%)
.bufr (1.4451%)	.grib (0.3977%)
.grb (1.4402%)	.bz2 (0.3083%)

Table 2: Statistics on ECFS tape storage.

The summary in Table 2 presents a total of 137.5 million files that are stored in 5.5 million directories and use 14.8 PB of capacity. The table also presents the most common file types by count and occupied capacity. Next to the unknown files that do not yield an extension on their file names, packed, compressed, and weather domain specific files are highly represented. Figure 2 shows a histogram of the most common file sizes. The system stores a large amount of files between 0 bytes and 1 KB, but else visually follows a Gaussian distribution that peaks at 8-16 MB.

The database excerpt also contains the *creation*, *modification*, and *read* timestamps of files. These timestamps mark the access times of a file on the tape drives and do therefore not reflect the access times of cached files. Files of the *Tiny* group (see Table 1), for example, are fully cached on disk and never retrieved from tape. The file system statistics of Table 2 show 101.3 million files that were never read from tape. These files were only uploaded or modified. If they were accessed, they were read from the HPSS' disk cache.

The upper graph of Figure 3 visualizes the absolute number of existing files at a particular point in time and the number of files that were *unread* or *unmodified* since

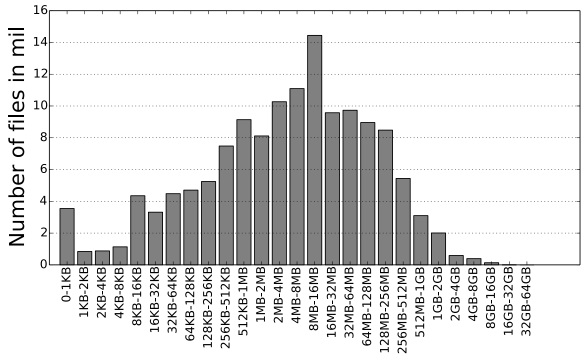


Figure 2: Histogram of stored ECFS files sizes.

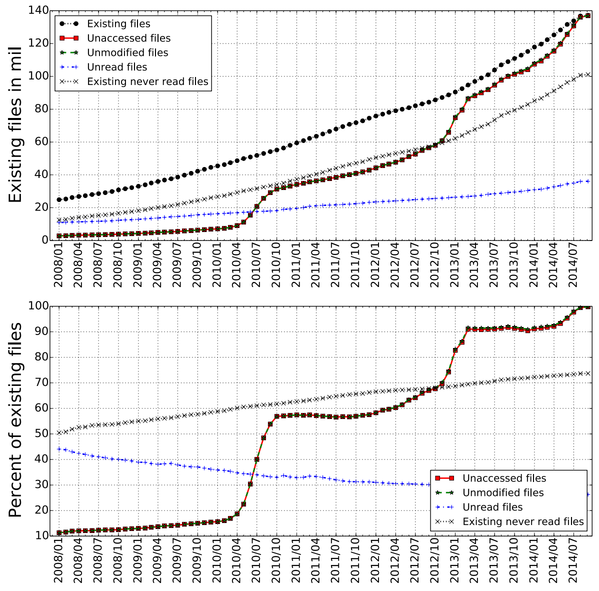


Figure 3: Top: Total number of existing files with amounts of unmodified and unread files since the point in time. Bottom: Fraction of unread or unmodified files relative to existing files.

that date. An *unaccessed file* was neither read nor modified and an *existing never read file* has no read time stamp and was therefore created and possibly updated only. The lower graph presents the fractions relative to the existing number of files to delineate the behavior over time. Since 2012/01 60 million new files were created. With the beginning of 2013, the new supercomputer becomes visible as more files are created and the graph steepens. In general, the number of existing files follows an exponential growth. The number of unmodified files closely follows the number of unaccessed files. This means that most of the last actions on files were modifications, which also includes the initial creation, and not reads. In the last year (2013/07 - 2014/07), the amount of never read files slightly grew from 68% to 73%, while the number of unread files shrank from 27% to 25%. In 2013/07 about 90% of the data stored on tape were neither read nor modified. This value changes by looking back an additional year till 2012/07 which covers the introduction of the new supercomputer. About 64% of the files existing at that point in time were not accessed until 2014/07.

4.2 Workload Characterization

For the time period from 2012/01/01 to 2014/05/20 a full trace of all ECFS operations has been investigated. Table 3 summarizes the key metrics.

Total GET requests	38.5 mil.
Total GET bytes	7.24 PB
Total PUT requests	78.3 mil.
Total PUT bytes	11.83 PB
Total DEL requests	4.2 mil.
Total RENAME requests	6.4 mil.
Total different files	73.4 mil.
Total different dirs	6.2 mil.
#Files with PUT	66.2 mil.
#Files with GET	12.2 mil.
Cache hit ratio by requests	86.7%
Cache hit ratio by bytes	45.9%

Table 3: Characterization of ECFS workload 2012/01/02 - 2014/05/21.

During the observed timespan, a total of 38.5 million GET requests were executed on 12.3 million unique files, a total of 78.3 million PUT requests on 66.2 million unique files were counted, and 4.2 million files were deleted from ECFS. As there are more unique written files than PUT requests, some files were updated or overwritten. In comparison to the NCAR analysis [14] where 30% of the stored files were read during the 29 month observation timespan, ECFS saw reads on 12.3 million distinct files which is 9% of the total corpus.

We analyzed the number of PUT and GET requests and their respective traffic based on the ECFS file size categories of Table 1. Figure 4 breaks down these metrics on a monthly basis. Until 2013/03, the system has a balanced throughput of 200-300 TB PUT and GET traffic per month with slightly prevailing write traffic. With the introduction of the new computer in the first quarter of 2013, the amount of written data doubles both, traffic- and request-wise, while the retrieval rates and volume merely rise.

Figure 5 visualizes the hotness of files during the observed timespan. As the analysis in Section 4.1 shows, the system contains a lot of data that have not been accessed within this time frame or were just uploaded without being retrieved again.

The main argument that is underlined by the plot is that in total 50% of all GET requests hit less than 5% of the stored files. This argument is supported by the overall 86.7% disk cache hit ratio with a disk-to-tape ratio of 1:43. The cached requests are responsible for

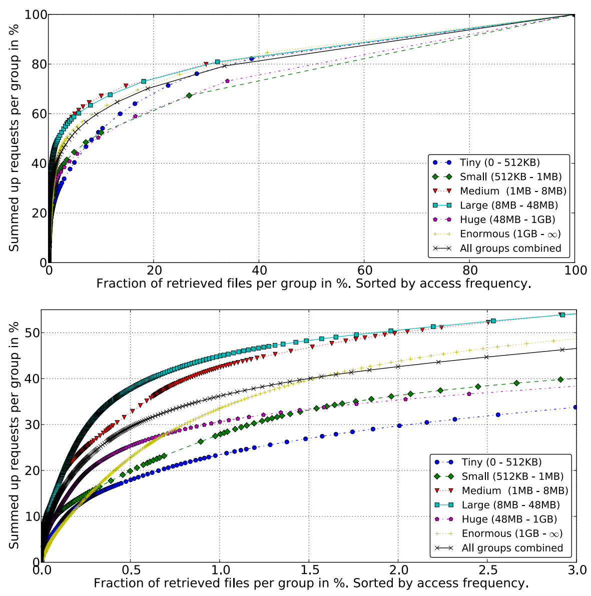


Figure 5: Top: CDF over file GET requests per file per size group.
Bottom: Zoomed to most frequently retrieved 3 %.

45.9% of the retrieved bytes, which leads to the assumption that most small reads can be satisfied by the cache and mostly larger files are retrieved from tape. The later cache analysis in Figure 15 (see Section 7) visualizes the cache hit ratios observed from the ECMWF traces for the different file size categories. While *Tiny* files achieve 100%, only 60% of the *Huge* and 50% of the *Enormous* file retrieval requests are served from disk.

4.3 User Session Analysis

For every request in the trace, the user id and the host that executed the command are known. A user id can be used by all sorts of processes running on multiple systems at the same time. The trace reveals 1,190 unique user ids and 2,647 unique hostnames. As presented in Table 3, a total of 11.8 PB of data have been written and 7.2 PB have been retrieved from the archive. Figure 6 visualizes how the bytes and requests are distributed to the identified users. The plot shows that only 850 users wrote data while 1,075 users retrieved data. Furthermore, only a small fraction of less than 100 users make up more than 90% of the traffic.

We gathered all commands issued by a user id from the same hostname and create clusters of executed requests that occurred within close succession. If the time between two requests is longer than the window, they are clustered into different groups. We call all requests within such a group a user session. Figure 7 presents the number of actions per identified user session for a growing time window based on the methodology used

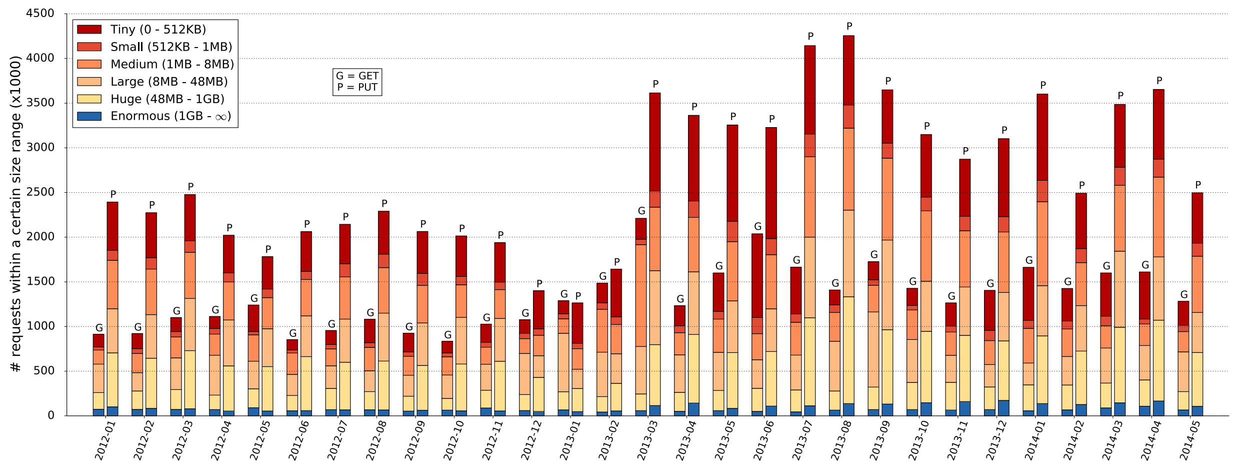


Figure 4: Total requests per month

Key	P05	P50	mean (+-95%)	P95	P99	Count
#Sessions per user_id	1	35	2,276.76 (± 1,006.85)	7,315	28,861	1,190
#Sessions per user_id@host	1	4	92.70 (± 7.50)	352	1,512	29,227
Total #Actions	2	7	47.04 (± 0.69)	126	579	2,709,343
GET Requests per session	1	4	35.55 (± 0.78)	108	571	1,083,067
ReGET requests per session	1	2	31.98 (± 3.66)	99	442	132,515
PUT Requests per session	1	5	34.43 (± 0.44)	97	373	2,274,645
Dirs with GETs	1	2	8.15 (± 0.16)	21	96	1,083,067
Dirs with PUTs	1	2	6.06 (± 0.07)	14	71	2,274,645
Retrieved files per directory	1	1.78	10.05 (± 0.23)	30	149.75	1,083,067
Archived files per directory	1	2	7.44 (± 0.12)	27	74	2,274,645
Retrieved MBytes	0.56	192.13	7,172.91 (± 221.74)	17,150.69	86,444.15	1,083,067
Archived MBytes	0.02	206.04	5,591.52 (± 94.84)	19,588.74	64,995.07	2,272,399
Session lifetime in s	0	154	2,601.60 (± 21.13)	9,295	38,456	2,709,343
Gap between sessions	120	250	896.26 (± 11.70)	3,070	3,500	29,227

Table 4: ECFS user session analysis. A total of 2,709,343 sessions were identified.

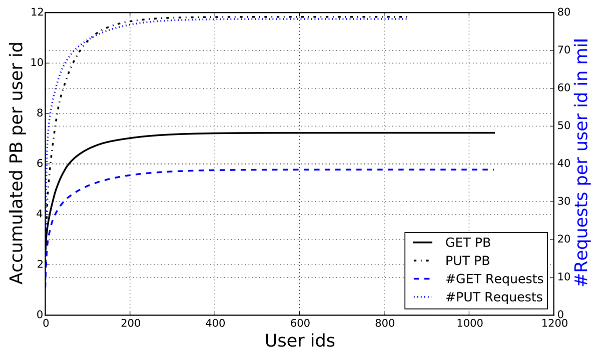


Figure 6: Summed up traffic for GET and PUT requests per unique user id

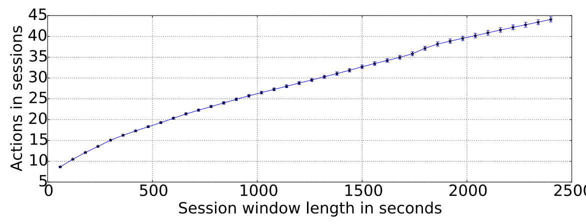


Figure 7: Mean actions per user sessions for growing window sizes.

by Madden et al. [21]. In contrast to Madden's approach, the graph does not show a plateau that would characterize a typical session. Therefore, we used a machine learning approach over all actions of each user to identify a window size that produces the most stable clusters for that user. For the 1,190 users, we identified a total of 2,709,343 sessions and Table 4 presents statistics over some key performance points. As for all following statistics, we present the 5, 50, 95 and 99 percentiles, a mean with a 95% confidence interval and the count of occurrences of the performance points. The statistics for a metric like *GET requests per session* is only counted if the session had at least one GET request. A write only session would not be counted here.

The results underline the trend shown in Figure 6. A small fraction of users is very active which becomes visible in the high differences of the P50 and P95 percentiles of multiple performance points. Also the volume in terms of requests and transferred bytes follows this trend, where a small fraction of sessions dominate the workload. The session lifetime considers the time from the beginning of the first until the end of the last action within a user session. Many sessions consist of a single action that is executed within a second or less. On the other hand, we observe sessions longer than 10 hours for the P99 which are most probably regular operations like cron jobs for backups.

The trace file yields full obfuscated paths of accessed files. Therefore, we investigated the locality of accesses within a session. If a session retrieved files, on average 36 GET requests were issued which accessed 8 directories with about 10 files per directory. If a session also archived data, on average 34 files were uploaded to 6 different directories with about 7 files per directory. As observed in the study by Adams et al. [1], we also see user sessions that re-retrieve the same file within the lifetime of the session, which occurred in 132,515 of the 1,083,067 sessions with GET requests. Out of the total 38.5 million GET requests, 11% (4.2 million) were re-retrievals of files within a user session.

5 MARS Database

The Meteorological Archival and Retrieval System (MARS) is the main repository of meteorological data at ECMWF. It contains petabytes of operational and research data, as well as data from special projects. In contrast to the ECFS, where files are identified by a unique path, MARS hosts billions of metereological fields that cannot be directly addressed by a user, but are the result of a query. The available log files of the MARS system contain all parameters of the queries and the number and source of the returned fields, but do not allow to identify the exact keys of the accessed fields. Therefore, this analysis cannot investigate the hotness of fields or files, but can only characterize the observed traffic.

MARS is based on a 3-tier storage architecture with the FDB as the first, the MARS servers' disk caches as the second and the HPSS tapes as the third layer. All fields in MARS are eventually persisted to the files in the HPSS tape backend, but requests are primarily served by the FDB and the MARS servers. The system also applies domain specific knowledge to improve the cache hit rates. For example, if files or full tapes are identified as hot, they can be manually loaded and locked to the MARS servers' disk caches. Currently 250 TB are reserved for this manual cache optimization.

5.1 Metadata Snapshot Analysis

The following analysis investigates an HPSS database snapshot of all MARS files on tape from the 2014/09/04. Table 5 presents a summary of the findings. Compared to ECFS, MARS stores a significantly smaller amount of files that use a larger total capacity of 37.9 PB. When the snapshot was taken, a total of 7.8 million of the 9.7 million stored files were never read from tape.

File system stats	
Total #files	9.7 mil.
Total used capacity	37.9 PB
Largest file size	1.34 TB
#Directories	555,799
Max files per directory	38,375
#Files never read from tape	7.9 mil. (24.9 PB)

Table 5: Statistics on MARS' tape storage.

Similar to ECFS, the histogram of the file sizes in Figure 8 resembles a Gaussian distribution, yet with a higher average file size and a higher maximum at 128-256 MB. The size of the largest file stored is 1.34 TB.

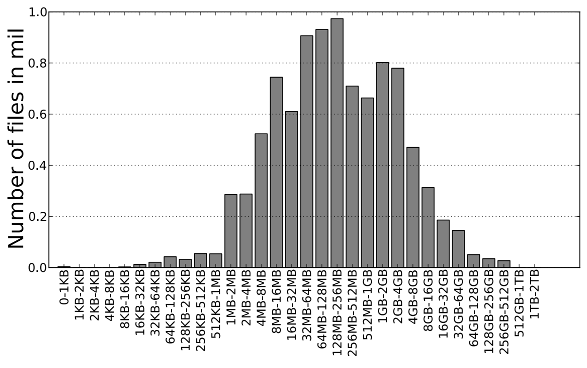


Figure 8: Histogram of files sizes

The visualization of *creation*, *modification*, and *read* times in Figure 9 follows the schematic described in Section 4.1. Again, the upper graph visualizes the absolute number of existing files at a particular point in time and the number files that have not been modified or read since that date. The lower graph presents the fractions relative to the existing number of files. The high rate of unaccessed files and a modification rate close to 100% shows that files are predominantly created, rarely updated and only read sometimes from the HPSS tape backend. The lower graph shows that up to 80% of the files on tape were written, but never read again. This behavior either indicates a cold storage or a strong caching infrastructure. In contrast to the ECFS analysis (see Figure 3), the introduction of the new supercomputer in Q1/2013 is not visible in Figure 9. Though Figure 10 shows a significant change of daily written fields and bytes, the file creation rate on the HPSS system does not change. This is because new fields are aggregated at the FDB and MARS server levels that are written as a file which then appears as a new file in HPSS. The assumption is that the average size of newly created files grows over time.

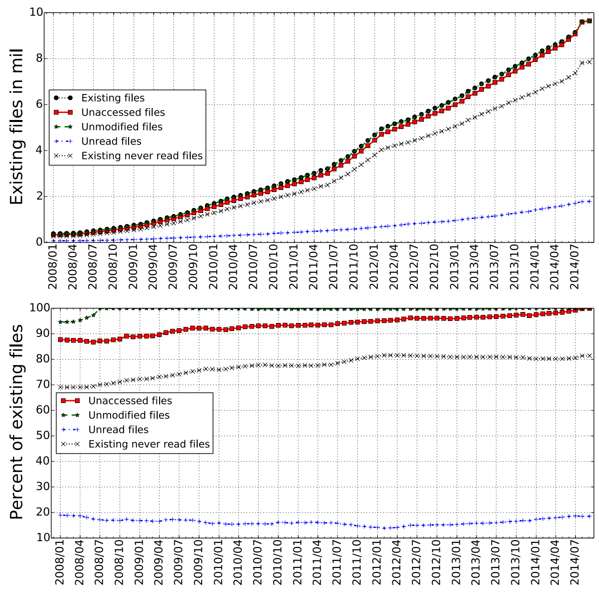


Figure 9: Top: Total number of existing files with amounts of unmodified and unread files since the point in time. Bottom: Fraction of unread or unmodified files relative to existing files.

5.2 Workload Characterization

We present the analysis of the MARS feedback logs over the timespan 2010/01/01-2014/02/27 that quantifies user requests and the resulting traffic. Figure 10 visualizes the daily throughput in bytes and requests. Again, the introduction of the new compute environment in Q1/2013 becomes visible in elevated throughput rates. The old environment can be characterized by a constant daily read rate of 40-50 TB (100-120 million fields) and 15-20 TB (about 50 million fields) of written data. With the new computer, the read rate doubles, but the write rate nearly increases threefoldly. During the peak throughput rate in 2013/01 both old and new supercomputers were running.

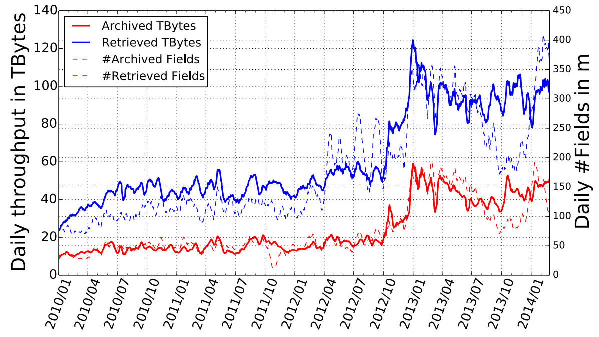


Figure 10: Throughput and number of accessed fields.

Table 6 presents some key characteristics of the considered 50 months. A total of 1.2 billion read requests were executed that fetched 269 billion fields which accounted for 91.6 PB of data, while 115 million requests created 114.7 billion new fields which account for 35.9 PB data. The logs breakdown each request into the number of fields, their summed up sizes and source of the returned fields. In total, 80.7% of all requests can be fully served by the field database (FDB), 17.6% of the requests also require data from the MARS server's disk drives and only 2.2% of the requests include data from tapes. 1.4% of the retrieve requests are fully served from tape without any cached data from disk. Considering the number of retrieved fields, MARS achieves a 95.1% cache hit ratio with a 1:38 disk-to-tape ratio on the MARS servers and a similar but not concretizable ratio on the FDB.

Total retrieved bytes (fields)	91.6 PB (269 bil.)
- from FDB bytes (fields)	54.2 PB (212 bil.)
- from MARS/disk bytes (fields)	29.4 PB (43.3 bil.)
- from HPSS/tape bytes (fields)	8 PB (13.3 bil.)
Total retrieve requests	1.2 bil.
- including FDB	992 mil. (85.3 %)
- from FDB only	938.9 mil. (80.7%)
- including MARS/disk	204.9 mil. (17.6%)
- from MARS/disk only	151.3 mil. (13%)
- including HPSS/tape	25.3 mil. (2.2%)
- from HPSS/tape only	16 mil. (1.4%)
Total archive requests	115 mil.
Total archived bytes (fields)	35.9 PB (114.7 bil.)

Table 6: Characterization of MARS workload 2010/01/01-2014/02/27.

Figure 11 characterizes the most active user ids in terms of retrieved and archived bytes and the according requests. It shows that the number of users who created content is significantly smaller than the number of users who retrieved data. A huge amount of traffic was actually generated by only two user ids. Some ids are shared by multi-purpose users, behind which multiple real user ids hide. Therefore, these very active users have to be considered as outliers.

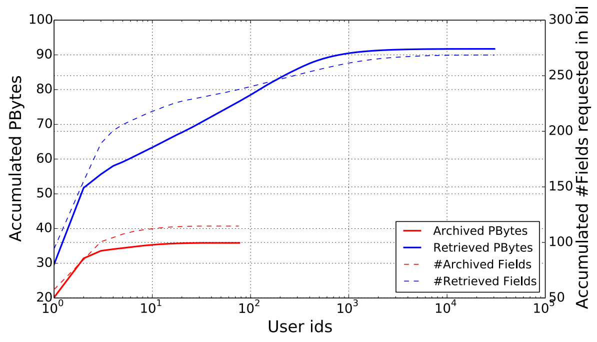


Figure 11: Traffic per unique user id.

6 Tape Mount Logs

Both ECFS and MARS use a HPSS powered tape archive as the final destination and primary copy of files. Despite the strong caching infrastructure, the tape robots are heavily used. Figure 12 illustrates the life cycle of

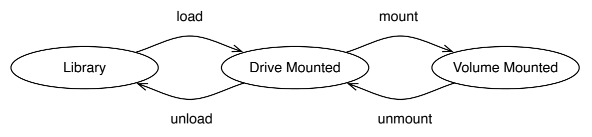


Figure 12: Tape states

one tape use in a simplified state diagram. To read data, a request is sent to the HPSS system that *loads* the tape into a drive. The tape is *mounted* as a volume and can be accessed. After a fixed timespan or on request, the volume is *unmounted*, but the tape can remain in the drive. If the tape is requested again while still being loaded, it can be *remounted*, or in the worst case *reloaded* into another drive. Eventually the tape is *unloaded* from the drive and put back to the library. Each tape has a unique identifier that indicates its type (STK T10k-B/C/D) and is assigned to ECFS or a MARS project. We use the identifiers to track the usage of the cartridges and map them to either ECFS or MARS. During the complete log period we saw a total of 231 different drive identifiers and 9,594 unique tape ids for ECFS and 23,118 for MARS.

Tape mount frequencies							
System	#Tapes	P05	P25	P50	mean (+-95%)	P95	P99
MARS	23,118	0	2	46	291.22 (± 9.70)	1,106	3,351
ECFS	9,594	1	12	85	296.64 (± 11.18)	1,408	2,470

Tape mount latencies in seconds							
System	#Mounts	P05	P25	P50	mean (+-95%)	P95	P99
MARS	6,730,218	26	30	35	54.35 (± 0.06)	155	262
ECFS	2,845,154	25	28	32	48.19 (± 0.07)	138	257

Table 7: Tape mount statistics

Figure 13 presents the access frequencies of tapes. Sorted by the most often accessed tapes, the total number of loads is summed up. The graph shows that MARS is accountable for 6.7 million and ECFS for 2.8 million tape loads. The right graph of the figure compares MARS and ECFS and reveals a similar distribution pattern. About 20% of all tapes are accountable for 80% of all mounts and more than 50% of the tapes are accessed in less than 5% of the loads.

Next we investigate the behavior of the tape system

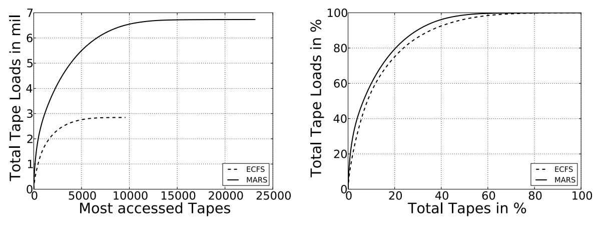


Figure 13: CDF over load requests per tape cartridge. Left: Absolute. Right: Normalized.

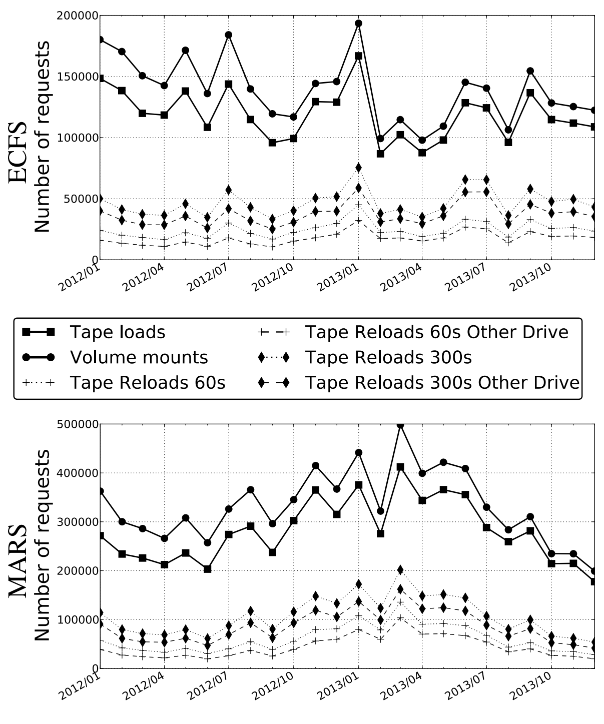


Figure 14: Mount Details for ECFS and MARS

over time. Both, the robot mount logs and the WHPSS logs were used for the analysis. Unfortunately, 6 days of the WHPSS logs were erroneous. We used the logs to feed finite state machines to track the cartridge states. Table 7 presents statistics of the time till a requested tape is available for work (volume mounted) and the number of load requests.

The ratio of 6.7 million MARS loads to the 2.8 million ECFS loads perfectly reflects the ratio of the stored data of 35.9 PB to 14.8 PB. The tape load times for the two categories are very similar, which leads to the assumption of equal or shared hardware in the backend. Although the median waiting time is only 35 (32) seconds, the mean is much higher due to some very long waiting times. More than 5% of all tape loads take more than 2 minutes and 1% of the loads take longer than 4 minutes.

Figure 14 visualizes the HPSS behavior for ECFS and MARS over time. The graphs show more volume mounts than tape loads, which shows the fraction of remounts without tape movements. The bottom of the graphs present the number of tape reloads and volume remounts within 60 and 300 seconds. A *tape reload 60s* means that after the tape was unloaded, within 60 seconds another mount request was issued. Over the observed time frame ECFS and MARS show a total of 21% of reloads within 60 seconds and 39% of reloads within 300 seconds. The significant finding is that in total 14.8% of all loaded tapes were unloaded from another drive less than 60 seconds ago.

Time slot (s)	MARS	ECFS
(0, 60]	73.922	8,740
(60, 120]	48,580	8,270
(120, 300]	107,276	23,126
(300, 600]	129,861	31,145
(0, 600]	359,639	71,281
(600, 1200]	189,691	46,792
(1200, 1800]	146,249	36,733
total success	695,569	154,806
[0, -1800]	849,229	162,965
(1800, fail]	8,061,777	1,364,171
Positive hit rate in %	7.24	9.20
Neutral hit rate in %	8.84	9.69
Total misses in %	83.92	81,11

Table 8: Prefetching hits based on the correlation analysis

6.1 Tape Prefetching

The average loading time could be improved by prefetching the tapes which are likely to be read next. In order to identify such tapes, we performed a correlation analysis on the tape mount logs using the Pearson correlation. In the following we describe the procedure and estimate the potential for improvement.

For any combination of two tapes *x* and *y* with a correlation coefficient of at least 0.8, we analyzed all load requests of *x* and measured the time difference until *y* was requested. Assuming the system would prefetch *y* once it sees a request for *x*, then this delay indicates the time *y* occupies a drive until it is requested. Table 8 shows the number of load requests within different time slots for the ECFS and MARS tapes.

We consider access delays of more than 1,800 seconds as prefetching failures, because with a high probability the tape would be evicted before being accessed. Furthermore, the interval [0, -1,800] shows the number of operations that did not result in a hit, but saw a load of *y* within the preceding 30 minutes. This is the case if *x* and *y* are requested at the same time, if *y* is already loaded or *y* was requested prior to *x*. In this case *x* would be the prefetching result of *y*'s load request and therefore, should not issue a prefetching event itself. This time slot neither generates any profit, nor induces any costs, which is why we call these events neutral hit and do not consider them as misses. Misses are load requests of *x* that never see a corresponding load event of *y* during the following 1,800 seconds.

The MARS and ECFS logs show a total of 695,569 and 154,806 prefetching hits which on average would have resulted in a hit every 74 seconds. Considering the mean latency of 54.35 and 48.19 seconds per load request, the latency of these operations would have accumulated to 1.20 and 0.24 years and could be saved by prefetching. The total load request latency of all load operation of the two projects is 11.60 and 4.35 years, respectively. The above mentioned reduction of 1.20 and 0.24 years could reduce these by 10.3% and 5.5%. This is a theoretical upper limit, since the 9.4 million misses would nearly double the amount of tape loads and clearly is unsuitable.

To design a prefetching strategy, possible candidates have to be identified. Furthermore, it has to be verified that the robots and drives have enough spare resources to process the prefetching load operations without impairing operational use. The logs show an average of 546.33 ($\pm$ 3.65) load operations per hour and during the busiest 5% of hours, more than 894 operations were performed. During the peak 1% utilization, more than 1,046 load operations were executed per hour. In the absence of such peak loads, the robots should be able to handle additional loads induced by prefetching misses. Finally, prefetching would not be applicable if the drives are constantly busy. We consider a drive to be idle if a tape is loaded but not mounted. Since mounted tapes are always loaded, the ratio of the volume mount time to the tape loaded time is a good metric for the drives utilization. We calculated this ratio for every hour of the observed time frame and on average see that tapes are accessed 82% of the time they are loaded. For the 0.01 and 0.99 percentiles we see a usage of 65% and 94%, respectively. This shows that the drives are highly utilized, but offer idle time to apply prefetching strategies.

7 Cache Simulation

Using a newly designed cache evaluation environment, different cache eviction strategies have been analyzed running the ECFS trace files (which were described in Section 4.2). We reuse the file size categorization of Table 1 and investigate the cache efficiency for different caching strategies and cache sizes. All GET, PUT, DEL, RENAME operations of the trace are replayed to a simulator that mimics a simple disk cache. A GET request on a non-existent file triggers a cache miss and the file is loaded to the cache. Also all PUT requests load the file into the cache, which might lead to an eviction. We evaluate the following cache eviction strategies using the ECFS traces that cover a timespan from 2012/01 till 2014/05 and are visualized in Figure 4.

LRU Data is evicted on a Least-Recently-Used strategy.

MRU Data is evicted on a Most-Recently-Used strategy.

FIFO Queue based eviction.

RANDOM A random cache entry is chosen for eviction. The presented graphs show the average results over 10 runs.

ARC Adaptive Replacement Cache that keeps track of both frequently and recently used files with an eviction history [22].

Bélády Adaption of the Bélády algorithm [6] which evicts those files that will not be needed for the longest time in the future. This algorithm would only be optimal if all files had the same size, but nevertheless we use this almost perfect cache as a baseline. The construction of an optimal cache is NP hard [9].

The results are visualized in Figure 15. For every file size category used at ECMWF (see Table 1) we present a graph that analyzes the cache hit ratio for the different caching strategies and multiple cache sizes. The last row shows the relative difference for a single *combined* cache for all files against the combined hit ratio over the sum of all hits and misses of the 6 subcaches. Its capacity steps are the sum of the same step of the other six caches. A negative result means that a single huge cache has a better hit ratio in terms of requests. We used the full year of 2012 to warm up the caches and present the total cache hit ratio for the period from 2013/01 to 2014/05.

Only the ECMWF baseline for *Tiny* files achieves a 100% hit ratio, as all files are always held on disk. The first GET request on a file that had no previous PUT request in the observed trace will result in a cache miss. Therefore, our results cannot reach 100%. The rightmost capacity of the graphs present a cache with unlimited size which never evict files because it can hold all of them. Therefore, this point presents the theoretical maximum cache hit ratio for the observed time frame.

The graphs show that strategies like *MRU* and *FIFO* are not usable at all. Only for very large caches, they yield an acceptable hit ratio. Due to the high number of re-GET requests, the most recently used files should be cached and not evicted, which explains the bad hit rates for *MRU*. Also the *FIFO* reveals a bad performance because it neglects the popularity of files. The constant writes of new files will evict files independent of their usage patterns.

While *MRU* and *FIFO* show a harmful behavior for cache efficiency, the *Random* strategy provides an unoptimized baseline. The *LRU* strategy accommodates the observations of the user sessions and the hot files presented in Figure 5 as it does not evict recently used files. The *ARC* cache competes as an improved LRU and in general slightly outperforms the *LRU* strategy.

The *Bélády* provides the best results, but as a theoretical construct that requires knowledge of the future we can only use it as an upper limit. Even for small cache sizes, this strategy often reaches the maximum hit ratio. This observation creates the assumption that an anticipatory eviction strategy that learns from the past might out-

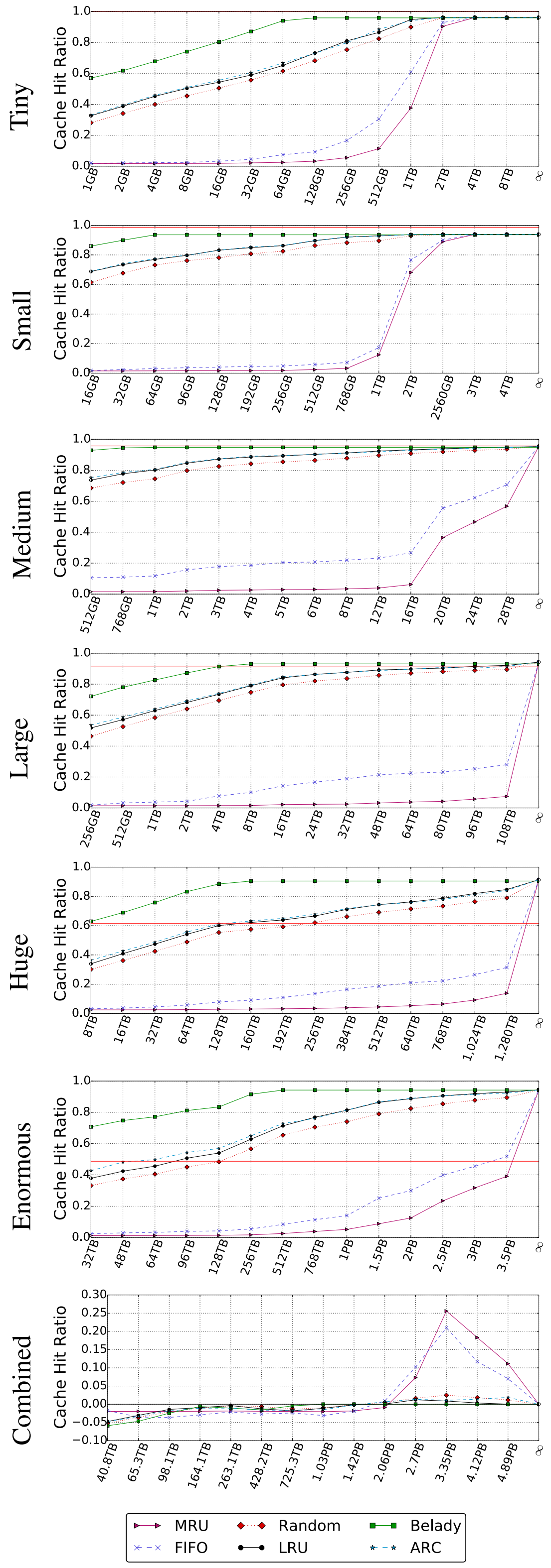


Figure 15: ECFS cache hit ratio evaluation. 2012 is used for cache warm up. Ratios are measured for 2013/2014. Horizontal red line marks ECMWF's hit ratio.

perform the other strategies. Domain knowledge as observable from the user sessions presented in Section 4.3 is available and could be fed to the caches.

Interestingly, the analysis of a big combined cache reveals that up to a cache size of about 2 PB, the single big instance in all cases provides a better hit rate. For a total cache size larger than 2 PB, the 6 subcaches provide better results for all cache strategies. All visualized hit ratios follow an upward trend for more capacity. The simulator can help to identify the achievable improvement of the hit ratio for extra cache capacity.

8 Discussion & Conclusion

This work analyzed log files and database snapshots to understand the behavior of ECFS and MARS, the two main archival systems at ECMWF, including the tape libraries that form the storage backbone of the two systems. The ECFS system resembles a typical archive and our findings underline the characterizations of previously studied systems [1, 21]. We analyzed the caching infrastructure of ECFS and provide a model and simulator to compare caches with different strategies and capacities. While ECMWF uses a lot of domain-specific knowledge which cannot be described algorithmically, we used the simulator to test basic strategies. It turned out that the *Adaptive Replacement Cache* (ARC) which evicts the *least recently used* and *least frequently used* files from the cache performs best. This conforms to our observation that files are often retrieved again after a short while (usually in the same user session). Coupling our test results with the knowledge of ECMWF will help in the future to further improve their cache hit rates.

Unlike ECFS, MARS opens the uncharted category of active archives that has not been thoroughly investigated until now. The object database uses a three-tiered architecture and a custom query language. All stored fields are subject to be read by computational models or experiments at any time. The investigated log files do not provide all the information necessary to fully characterize the user traffic. It is, for example, not possible to deduce the exact fields returned, although we know the queries. Nevertheless, it was possible to roughly describe the traffic and to assess the cache efficiency and the usage of the tape backend.

For ECFS, we saw read accesses on only 9% of the file corpus with a disk cache hit ratio of 86.7% during the observed timespan. Only 26,3% of the files on tape were ever read. The MARS logs do not reveal deeper information about which files and fields were accessed, but show that 95.1% of the requested fields were served from disk caches where only 2.2% of all requests included accesses to one or multiple tapes. Despite of this strong caching infrastructure, we have observed more than 9.5 million tape loads over a period of two years with 5% of the tapes being loaded more than 1,000 times.

The archival system's latency is not the primary bottleneck for the computations, as most operations run in batch and wait until the requested data is available. Nevertheless, since extensive queries that involve tapes can take several minutes, hours or even days, it is worthwhile to improve the average loading time of the tapes. Although the system is already well-configured, the 60 second tape reload rate of 14.8% and the prefetching analysis expose further potential for optimization.

The quality of weather forecasts constantly improves due to faster computers and improved computational models. At the same time, the requirements for storage infrastructure grow as well. Kryder's Law states that the areal density of bits on disk platters roughly doubles every two years [27]. While this was true for the last three decades, Rosenthal et al. forecast that the improvements in storage cost per bit for disk and tape will be much slower [24]. ECMWF faced a CAGR of 45% over the last years, which lately increased to 53% with the latest supercomputer. While this growth was maintainable with a constant budget during the last years, it might lead to the economical threats for long-term digital storage described by Baker et al. [5]. The consequence has to be an adjustment of scenario planning or implementing means to grow slower.

Due to a lack of studies, it is difficult to compare or generalize our results to other archival storage environments. Nevertheless, we believe that they are also relevant for other existing or future systems and that they can help to make the right design decisions. The reason is the observation that many large systems share at least some essential properties like the small read-to-write ratio and the overall architecture combining a large tape library with a relatively small disk-based cache. We developed a generic, extensible set of tools that can be applied to analyze workloads of archives and data centers. The cache simulation, for instance, helps to evaluate new caching strategies and to explore the impact of different eviction strategies and cache bucket sizes.

9 Closing Remarks

We are very grateful to the European Centre for Medium-Range Weather Forecasts for the opportunity to browse and analyze their log files and to share their valuable knowledge on building large scale archival systems.

The analyzed traces are stored at ECMWF and will be made available upon individual requests. The cache simulator, part of the scripts, and links to the trace files are available at https://github.com/zdvresearch/fast15-paper-extras.

References

[1] ADAMS, I., STORER, M., AND MILLER, E. Analysis of Workload Behavior in Scientific and Historical Long-Term Data Repositories. *ACM Transactions on Storage (TOS)* (May 2012).

[2] AGRAWAL, N., BOLOSKY, W., DOUCEUR, J., AND LORCH, J. A Five-year Study of File-system Metadata. *ACM Transactions on Storage (TOS)* (Oct. 2007).

[3] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload Analysis of a Large-scale Key-value Store. In *Proc. of the 12th ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)* (2012).

[4] BAKER, M., KEETON, K., AND MARTIN, S. Why Traditional Storage Systems Don't Help Us Save Stuff Forever. In *Proc. of the 1st Workshop on Hot Topics in System Dependability (HotDep)* (2005).

[5] BAKER, M., SHAH, M., ROSENTHAL, D., ROUSSOPOULOS, M., MANIATIS, P., GIULI, T., AND BUNGALE, P. A Fresh Look at the Reliability of Long-term Digital Storage. In *Proc. of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)* (2006).

[6] BÉLÁDY, L. A Study of Replacement Algorithms for a Virtual-storage Computer. *IBM Systems Journal* (June 1966).

[7] BRESLAU, L., CAO, P., FAN, L., PHILLIPS, G., AND SHENKER, S. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proc. of the 18th IEEE International Conference on Computer and Communications (INFOCOM)* (1999).

[8] CHEN, Y., SRINIVASAN, K., GOODSON, G., AND KATZ, R. Design Implications for Enterprise Storage Systems via Multi-Dimensional Trace Analysis. In *Proc. of the 23rd ACM Symposium on Operating Systems Principles (SOSP)* (2011).

[9] CHROBAK, M., WOEGINGER, G., MAKINO, K., AND XU, H. Caching Is Hard – Even in the Fault Model. In *Algorithms – ESA 2010*, M. de Berg and U. Meyer, Eds., vol. 6346 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2010.

[10] COLARELLI, D., AND GRUNWALD, D. Massive Arrays of Idle Disks for Storage Archives. In *Proc. of the ACM/IEEE conference on Supercomputing (SC)* (2002).

[11] DOUCEUR, J., AND BOLOSKY, W. A large-scale study of file-system contents. In *Proc. of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)* (1999).

[12] ECMWF. ECMWF Data Handling System. http://www.ecmwf.int/en/computing/our-facilities/data-handling-system, Sept. 2014.

[13] EVANS, K., AND KUENNING, G. A Study of Irregularities in File-Size Distributions. In *Proc. of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)* (2002).

[14] FRANK, J., MILLER, E., AND ROSENTHAL, I. A. D. Evolutionary trends in a supercomputing tertiary storage environment. In *Proc. of the 20th IEEE International Symposium on Modeling, Analysis, and Simulation (MASCOTS)* (2012).

[15] GILL, P., ARLITT, M., LI, Z., AND MAHANTI, A. Youtube Traffic Characterization: A View from the Edge. In *Proc. of the 7th ACM SIGCOMM Conference on Internet Measurement (IMC)* (2007).

[16] GRAWINKEL, M., BEST, G., SPLIETKER, M., AND BRINKMANN, A. LoneStar Stack: Architecture of a Disk-Based Archival System. In *Proc. of the 9th IEEE International Conference on Networking, Architecture, and Storage (NAS)* (2014).

[17] HUANG, Q., BIRMAN, K., VAN RENESSE, R., LLOYD, W., KUMAR, S., AND LI, H. C. An Analysis of Facebook Photo Caching. In *Proc. of the 24th ACM Symposium on Operating Systems Principles (SOSP)* (2013).

[18] JAFFE, E., AND KIRKPATRICK, S. Architecture of the Internet Archive. In *Proc. of the ACM Israeli Experimental Systems Conference (SYSTOR)* (2009).

[19] LEUNG, A., PASUPATHY, S., AND MILLER, G. G. E. Measurement and Analysis of Large-scale Network File System Workloads. In *Proc. of the Annual Technical Conference (ATC)* (2008).

[20] LOS ALAMOS NATIONAL LABORATORY. Archive Data to Support and Enable Computer Science Research. http://institutes.lanl.gov/data/archive-data/, 2014.

[21] MADDEN, B., ADAMS, I., FRANK, J., AND MILLER, E. Analyzing User Behavior: A Trace Analysis of the NCAR Archival Storage System. Tech. Rep. UCSC-SSRC-ssrctr-12-02, University of California, Santa Cruz, Mar. 2012.

[22] MEGIDDO, N., AND MODHA, D. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proc. of the USENIX Conference on File and Storage Technologies (FAST)* (2003).

[23] MEISTER, D., KAISER, J., BRINKMANN, A., CORTES, T., KUHN, M., AND KUNKEL, J. A study on data deduplication in hpc storage systems. In *Proc. of the ACM/IEEE Conference on Supercomputing (SC)* (2012).

[24] ROSENTHAL, D., ROSENTHAL, D., MILLER, E., ADAMS, I., STORER, M., AND ZADOK, E. The Economics of Long-Term Digital Storage. In *The Memory of the World in the Digital age: Digitization and Preservation* (2012), United Nations Educational, Scientific and Cultural Organization (UNESCO).

[25] SAROIU, S., GUMMADI, P., DUNN, R., GRIBBLE, S., AND LEVY, H. An Analysis of Internet Content Delivery Systems. In *Proc. of the 5th Conference on Operating Systems Design and Implementation (OSDI)* (2002).

[26] STORER, M. W., GREENAN, K. M., MILLER, E. L., AND VORUGANTI, K. Pergamum: Replacing Tape with Energy Efficient, Reliable, Disk-Based Archival Storage. In *Proc. of the 6th USENIX Conference on File and Storage Technologies (FAST)* (2008).

[27] WALTER, C. Kryder's Law. *Scientific American*, 293 (2005).

[28] WELCH, B., AND NOER, G. Optimizing a hybrid SSD/HDD HPC storage system based on file size distributions. In *Proc. of the 29th IEEE Conference on Mass Storage Systems and Technologies (MSST)* (2013).

Efficient Intra-Operating System Protection Against Harmful DMAs

Moshe Malka Nadav Amit Dan Tsafrir
Technion – Israel Institute of Technology

Abstract

Operating systems can defend themselves against misbehaving I/O devices and drivers by employing intra-OS protection. With "strict" intra-OS protection, the OS uses the IOMMU to map each DMA buffer immediately before the DMA occurs and to unmap it immediately after. Strict protection is costly due to IOMMU-related hardware overheads, motivating "deferred" intra-OS protection, which trades off some safety for performance.

We investigate the Linux intra-OS protection mapping layer and discover that hardware overheads are not exclusively to blame for its high cost. Rather, the cost is amplified by the I/O virtual address (IOVA) allocator, which regularly induces linear complexity. We find that the nature of IOVA allocation requests is inherently simple and constrained due to the manner by which I/O devices are used, allowing us to deliver constant time complexity with a compact, easy-to-implement optimization. Our optimization improves the throughput of standard benchmarks by up to 5.5x. It delivers strict protection with performance comparable to that of the baseline deferred protection.

To generalize our case that OSes drive the IOMMU with suboptimal software, we additionally investigate the FreeBSD mapping layer and obtain similar findings.

1 Introduction

The role that the I/O memory management unit (IOMMU) plays for I/O devices is similar to the role that the regular memory management unit (MMU) plays for processes. Processes typically access the memory using virtual addresses translated to physical addresses by the MMU. Likewise, I/O devices commonly access the memory via direct memory access operations (DMAs) associated with I/O virtual addresses (IOVAs), which are translated to physical addresses by the IOMMU. Both hardware units are implemented similarly with a page table hierarchy that the operating system (OS) maintains and the hardware walks upon an (IO)TLB miss.

The IOMMU can provide *inter-* and *intra-OS protection* [4, 44, 54, 57, 59]. Inter protection is applicable in virtual setups. It allows for "direct I/O", where the host assigns a device directly to a guest virtual machine (VM) for its exclusive use, largely removing itself from the guest's I/O path and thus improving its performance [27, 42]. In this mode, the VM directly programs device DMAs using its notion of (guest) "physical" addresses. The host uses the IOMMU to redirect these accesses to where the VM memory truly resides, thus protecting its own memory and the memory of the other VMs. With inter protection, IOVAs are mapped to physical memory locations infrequently, only upon such events as VM creation and migration, and host management operations such as memory swapping, deduplication, and NUMA migration. Such mappings are therefore denoted *persistent* or *static* [57].

Intra-OS protection allows the OS to defend against errant/malicious devices and buggy drivers, which account for most OS failures [19, 49]. Drivers/devices can initiate/perform DMAs to arbitrary memory locations, and IOMMUs allow OSes to protect themselves by restricting these DMAs to specific physical memory locations. Intra-OS protection is applicable in: (1) non-virtual setups where the OS has direct control over the IOMMU, and in (2) virtual setups where IOMMU functionality is exposed to VMs via paravirtualization [12, 42, 48, 57], full emulation [4], or, recently, hardware support for nested IOMMU translation [2, 36]. In this mode, IOVA (un)mappings are frequent and occur within the I/O critical path. The OS programs DMAs using IOVAs rather than physical addresses, such that each DMA is preceded and followed by the mapping and unmapping of the associated IOVA to the physical address it represents [38, 46]. For this reason, such mappings are denoted *single-use* or *dynamic* [16]. The context of this paper is intra-OS protection (§2).

To do its job, the intra-OS protection mapping layer must allocate IOVA values: integer ranges that serve as page identifiers. IOVA allocation is similar to regular memory allocation. But it is different enough to merit its own allocator (§3). One key difference is that regular allocators dedicate much effort to preserving locality and to combating fragmentation, whereas the IOVA allocator disallows locality and enjoys a naturally "unfragmented" workload. This difference makes the IOVA allocator 1–2 orders of magnitude smaller in terms of lines of code.

Another difference is that, by default, the IOVA subsystem trades off some safety for performance. It delays the completion of IOVA deallocations while letting the OS believe that the deallocations have been processed. Specifically, freeing an IOVA implies purging it from the IOTLB such that the associated physical buffer is no longer accessible to the I/O device. But invalidating IOTLB entries is a costly, slow operation. So the IOVA subsystem opts for batching the invalidations until enough accumulate and then invalidating all the IOTLB en masse, thus reducing the amortized price. This default mode is called *deferred protection*. Users can turn it off at boot time by instructing the kernel to use *strict protection*.

The activity that stresses the IOVA mapping layer is associated with I/O devices that employ *ring buffers* in order to communicate with their OS drivers in a producer-consumer manner. A ring is a cyclic array whose entries correspond to DMA requests that the driver initiates and the device fulfills. Ring entries contain IOVAs that the mapping layer allocates/frees before/after the associated DMAs are processed by the device. We carefully analyze the performance of the IOVA mapping layer and find that its allocation scheme is efficient despite its simplicity, but only if the device is associated with a single ring (§4).

Devices, however, often employ more rings, in which case our analysis indicates that the IOVA allocator seriously degrades the performance (§5). We study this deficiency and find that its root cause is a pathology we call *long-lasting ring interference*. The pathology occurs when I/O asynchrony prompts an event that causes the allocator to migrate an IOVA from one ring to another, henceforth repetitively destroying the contiguity of the ring's I/O space upon which the allocator relies for efficiency. We conjecture that this harmful effect remained hidden thus far because of the well-known slowness associated with manipulating the IOMMU. The hardware took most of the blame for the high price of intra-OS protection even though software is equally guilty, as it turns out, in both OSes that we checked (Linux and FreeBSD).

We address the problem by adding the Efficient IOVA allocatoR (EIOVAR) optimization to the kernel's mapping subsystem (§6). EIOVAR exploits the fact that its workload is (1) exclusively comprised of power-of-two allocations, and is (2) ring-induced, so the difference D between the cumulative number of allocation and deallocation requests at any given time is proportional to the ring size, which is relatively small. EIOVAR is accordingly a simple, thin layer on top of the baseline IOVA allocator that proxies all (de)allocations. It caches all freed ranges and reuses them to quickly satisfy subsequent allocations. It is successful because the requests are similar. It is frugal with memory because D is small. And it is compact (implementation-wise) because it consists of an array of freelists with a bit of minimal logic. EIOVAR entirely eliminates the baseline allocator's aforementioned reliance on I/O space contiguity, ensuring all (de)allocations are efficient.

We evaluate the performance of EIOVAR using different I/O devices (§7). On average, EIOVAR satisfies (de)allocations in about 100 cycles. It improves the throughput of Netperf, Apache, and Memcached benchmarks by up to 5.50x and 1.71x for strict and deferred protection, respectively, and it reduces the CPU consumption by up to 0.53x. Interestingly, EIOVAR delivers strict protection with performance that is similar to that of the baseline system when employing deferred protection.

Accelerating allocation (of IOVAs in our case) using freelists is a well-known technique commonly utilized by memory allocators [13, 14, 15, 29, 40, 53, 55] (§8). Our additional contributions are: identifying that the performance of the IOMMU mapping layer can be dramatically improved by employing this technique across the OSes we tested and thus refuting the common wisdom that the poor performance is largely due to the hardware slowness; carefully studying the IOMMU mapping layer workload; finding that it is very "well behaved"; which ensures that even our simplistic EIOVAR freelist provides fast, constant-time IOVA allocation while remaining compact in size (§9).

2 Intra-OS Protection

DMA refers to the ability of I/O devices to read from or write to the main memory without CPU involvement. It is a heavily used mechanism, as it frees the CPU to continue to do work between the time it programs the DMA until the time the associated data is sent or received. As noted, drivers of devices that stress the IOVA mapping layer initiate DMA operations via a *ring buffer*, which is a circular array in main memory that constitutes a shared data structure between the driver and its device. Each entry in the ring contains a DMA *descriptor*, specifying the address(es) and size(s) of the corresponding *target buffer(s)*; the I/O device will write/read the data to/from the latter, at which point it will trigger an interrupt to let the OS know that the DMA has completed. (Interrupts are coalesced if their rate is high.) I/O device are commonly associated with more than one ring, e.g., a receive ring denoted *Rx* for DMA read operations, and a transmit ring denoted *Tx* for DMA write operations.

In the past, I/O devices used physical addresses in order to access main memory, namely, each DMA descriptor contained a physical address of its target buffer. Such unmediated DMA activity directed at the memory makes the system vulnerable to rogue devices performing errant or malicious DMAs [9, 17, 38, 58], or to buggy drivers that might program their devices to overwrite any part of the system memory [8, 30, 42, 49, 56]. Subsequently, all major chip vendors introduced IOMMUs [2, 7, 34, 36], alleviating the problem as follows.

The OS associates each DMA target buffer with some IOVA, used instead of the physical address when filling out the associated ring descriptor. The I/O device is oblivious to the change, processing the DMA as if the IOVA was a physical memory address. The IOMMU then translates the IOVA, routing the operation to the appropriate

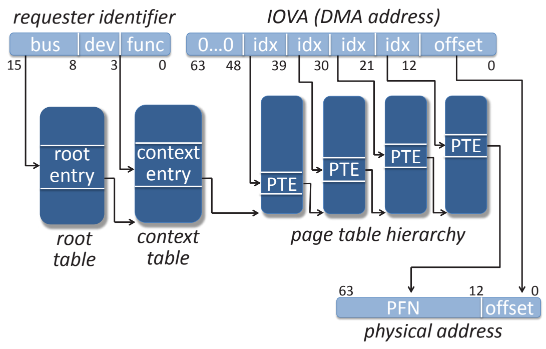


Figure 1: *IOVA translation using the Intel IOMMU.*

memory location. Figure 1 depicts the translation process of the Intel IOMMU used in this paper. The PCI protocol dictates that each DMA operation is associated with a 16 bit *request identifier* comprised of a *bus-device-function* triplet unique to the corresponding I/O device. The IOMMU uses the 8 bit bus number to index the *root table*, retrieving the physical address of the *context table*. It then indexes the latter using the device-function 8 bit concatenation, yielding the physical location of the root of the page table hierarchy that houses the device's IOVA translations. Similarly to the MMU, the IOMMU accelerates translations using an IOTLB.

The functionality of the IOMMU is equivalent to that of the regular MMU. It permits IOVA memory accesses to go through only if the OS previously inserted matching translations. The OS can thus protect itself by allowing a device to access a target buffer just before the corresponding DMA occurs (add mapping), and by revoking access just after (remove mapping), exerting fine-grained control over what portions of memory may be used in I/O transactions at any given time. This state-of-the-art strategy of IOMMU-based protection was termed *intra-OS protection* by Willmann et al. [57].It is recommended by hardware vendors [31, 38], and it is used by operating systems [6, 16, 35, 45]. For example, the DMA API of Linux—which we use in this study—notes that "DMA addresses should be mapped only for the time they are actually used and unmapped after the DMA transfer" [46].

3 IOVA vs. Memory Allocation

The task of generating IOVAs—namely, the actual integer numbers that the OS assigns to descriptors and the devices then use—is similar to regular memory allocation. But it is sufficiently different to merit its own allocator, because it optimizes for different objectives, and because it is required to make different tradeoffs, as follows.

Locality Memory allocators spend much effort in trying to (re)allocate memory chunks in a way that maximizes reuse of TLB entries and cached content. The IOVA mapping layer of the OS does the opposite. The numbers it allocates correspond to whole pages, and they are not allowed to stay warm in hardware caches in between allocations. Rather, they must be purged from the IOTLB and from the page table hierarchy immediately after the DMA completes. Moreover, while purging an IOVA, the mapping layer must flush each cache line that it modifies in the hierarchy, as the IOMMU and CPU do not reside in the same coherence domain.[1]

Fragmentation Memory allocators invest much effort in combating fragmentation, attempting to eliminate unused memory "holes" and utilize the memory they have before requesting the system for more. As we further discuss in §5–§6, it is trivial for the IOVA mapping layer to avoid fragmentation due to the simple workload that it services, which exclusively consists of requests whose size is a power of two number of pages. The IOMMU driver rounds up all IOVA range requests to 2^j for two reasons. First, because IOTLB invalidation of 2^j ranges is faster [36, 39]. And second, because the allocated IOVA range does not correspond to 2^j pages of real memory. Rather it merely corresponds to to a pair of integers marking the beginning and end of the range. Namely, the IOMMU driver maps only the physical pages it was given, but it reserves a bigger IOVA range so as to make the subsequent associated IOTLB invalidation speedier. It can thus afford to be "wasteful". (In our experiments, the value of j was overwhelmingly 0. Namely, the allocated IOVA ranges almost always consist of one page only.)

Complexity Simplicity and compactness matter and are valued within the kernel. Not having to worry about locality and fragmentation while enjoying a simple workload, the mapping layer allocation scheme is significantly simpler than regular memory allocators. In Linux, it is comprised of only a few hundred lines of codes instead of thousands [40, 41] or tens of thousands [13, 32].

Safety & Performance Assume a thread T_0 frees a memory chunk M, and then another thread T_1 allocates memory. A memory allocator may give M to T_1, but only after it processes the free of T_0. Namely, it would never allow T_0 and T_1 to use M together. Conversely, the IOVA mapping layer purposely allows T_0 (the device) and T_1 (the OS) to access M simultaneously for a short period of time. The reason: invalidation of IOTLB entries is costly [4, 57]. Therefore, by default, the mapping layer trades off safety for performance by (1) accumulating up to W unprocessed 'free' operations and only then (2) freeing those W IOVAs and (3) invalidating the entire IOTLB en masse. Consequently, target buffers are actively being used by the OS while the device might still access them through

[1]Intel IOMMU specification documents a capability bit that indicates whether the IOMMU and CPU coherence could be turned on [36], but we do not own such hardware and believe it is not yet common.

stale IOTLB entries. This weakened safety mode is called *deferred protection*. Users can instead employ *strict protection*—which processes invalidations immediately—by setting a kernel command line parameter.

Metadata Memory allocators typically use the memory that their clients (de)allocate to store their metadata. For example, by inlining the size of an allocated area just before the pointer that is returned to the client. Or by using linked lists of free objects whose "next" pointers are kept within the areas the comprise the lists. The IOVA mapping layer cannot do that, because the IOVAs that it invents are pointers to memory that is used by some other entity (the device or the OS). An IOVA is just an *additional* identifier for a page, which the mapping layer does not own.

Pointer Values Memory allocators running on 64-bit machines typically use native 64-bit pointers. The IOVA mapping layer prefers to use 32-bit IOVAs, as utilizing 64-bit addresses for DMA would force a slower, dual address cycle on the PCI bus [16].

4 Supposed O(1) Complexity of Baseline

In accordance to §3, the allocation scheme employed by the Linux/x86 IOVA mapping layer is different than, and independent of, the regular kernel memory allocation subsystem. The underlying data structure of the IOVA allocator is the generic Linux kernel red-black tree. The elements of the tree are *ranges*. A range is a pair of integer numbers $[L,H]$ that represent a sequence of currently allocated I/O virtual page numbers $L, L+1, ..., H-1, H$, such that $L \leq H$ stand for "low" and "high", respectively. Ranges are pairwise disjoint, namely, given two ranges $[L_1,H_1] \neq [L_2,H_2]$, then either $H_1 < L_2$ or $H_2 < L_1$.

Newly requested IOVA integers are allocated by scanning the tree right-to-left from the highest possible value downwards towards zero in search for a gap that can accommodate the requested range size. The allocation scheme attempts and—as we will later see—ordinarily succeeds to allocate the new range from within the highest gap available in the tree.

The allocator begins to scan the tree from a *cache node* that it maintains, denoted C. The allocator iterates from C through the ranges in a descending manner until a suitable gap is found. C is maintained such that it usually points to a range that is higher than (to the right of) the highest free gap, as follows. When (1) a range R is freed and C currently points to a range lower than R, then C is updated to point to R's successor. And (2) when a new range Q is allocated, then C is updated to point to Q; if Q was the highest free gap prior to its allocation, then C still points higher than the highest free gap after this allocation.

Figure 2 lists the pseudo code of the IOVA allocation

```
struct range_t {int lo, hi;};

range_t alloc_iova(rbtree_t t, int rngsiz) {

  range_t  new_range;
  rbnode_t right = t.cache;
  rbnode_t left  = rb_prev( right );

  while(right.range.lo - left.range.hi <= rngsiz)
      right = left;
      left  = rb_prev( left );

  new_range.hi = right.lo - 1;
  new_range.lo = right.lo - rngsiz;
  t.cache      = rb_insert( t, new_range );

  return new_range;
}

void free_iova(rbtree_t t, rbnode_t d) {
  if( d.range.lo >= t.cache.range.lo )
      t.cache = rb_next( d );
  rb_erase( t, d );
}
```

Figure 2: *Pseudo code of the baseline IOVA allocation scheme. The functions* rb_next *and* rb_prev *return the successor and predecessor of the node they receive, respectively.*

scheme as was just described. Clearly, the algorithm's worst case complexity is linear due to the 'while' loop that scans previously allocated ranges beginning at the cache node C. But when factoring in the actual workload that this algorithm services, the situation is not so bleak: the complexity turns out to actually be constant rather than linear (at least conceptually).

Recall that the workload is commonly induced by a circular ring buffer, whereby IOVAs of DMA target buffers are allocated and freed in a repeated, cyclic manner. Consider, for example, an Ethernet NIC with a Rx ring of size n, ready to receive packets. Assume the NIC initially allocates n target buffers, each big enough to hold one packet (1500 bytes). The NIC then maps the buffers to n newly allocated, consecutive IOVAs with which it populates the ring descriptors. Assume that the IOVAs are $n, n-1, ..., 2, 1$. (The series is descending as IOVAs are allocated from highest to lowest.) The first mapped IOVA is n, so the NIC stores the first received packet in the memory pointed to by n, and it triggers an interrupt to let the OS know that it needs to handle the packet.

Upon handling the interrupt, the OS first unmaps the corresponding IOVA, purging it from the IOTLB and IOMMU page table to prevent the device from accessing the associated target buffer (assuming strict protection). The unmap frees IOVA=n, thus updating C to point to n's successor in the red-black tree (free_iova in Figure 2).

The OS then immediately re-arms the ring descriptor for future packets, allocating a new target buffer and associating it with a newly allocated IOVA. The latter will be n, and it will be allocated in constant time, as C points to n's immediate successor (alloc_iova in Figure 2). The same scenario will cyclically repeat itself for $n-1, n-2, \ldots, 1$ and then again $n, \ldots, 1$ and so on as long as the NIC is operational.

Our soon to be described experiments across multiple devices and workloads indicate that the above description is fairly accurate. IOVA allocations requests are overwhelmingly for one page ranges ($H = L$), and the freed IOVAs are indeed re-allocated shortly after being freed, enabling, in principle, the allocator in Figure 2 to operate in constant time as described. But the algorithm succeeds to operate in this ideal manner only for some bounded time. We find that, inevitably, an event occurs and ruins this ideality thereafter.

5 Long-Lasting Ring Interference

The above $O(1)$ algorithm description assumes there exists only one ring in the I/O virtual address space. In reality, however, there are often two or more, for example, the Rx and Tx receive and transmit rings. Nonetheless, even when servicing multiple rings, the IOVA allocator provides constant time allocation in many cases, so long as each ring's free_iova is immediately followed by a matching alloc_iova for the same ring (the common case). Allocating for one ring and then another indeed causes linear IOVA searches due to how the cache node C is maintained. But large bursts of I/O activity flowing in one direction still enjoy constant allocation time.

The aforementioned event that forever eliminates the allocator's ability to accommodate large I/O bursts with constant time occurs when a free-allocate pair of one ring is interleaved with that of another. Then, an IOVA from one ring is mapped to another, ruining the contiguity of the ring's I/O virtual address. Henceforth, every cycle of n allocations would involve one linear search prompted whenever the noncontiguous IOVA is freed and reallocated. We call this pathology *long-lasting ring interference* and note that its harmful effect increases as additional inter-ring free-allocate interleavings occur.

Table 1 illustrates the pathology. Assume that a server mostly receives data and occasionally transmits. Suppose that Rx activity triggers a Rx.free_iova(L) of address L **(1)**. Typically, this action would be followed by Rx.alloc_iova, which would then return L **(2)**. But sometimes a Tx operation sneaks in between. If this Tx operation is Tx.free_iova(H) such that $H > L$ **(3)**, then the allocator would update the cache node C to point to H's successor **(4)**. The next Rx.alloc_iova would be satisfied by H **(5)**, but then the subsequent Rx.alloc_iova would have

operation	without Tx			with Tx		
	return value	C before	C after	return value	C before	C after
Rx.free(L=151) **(1)**		152	152		152	152
Tx.free(H=300) **(3)**					152	**(4)** 301
Rx.alloc	**(2)** 151	152	151	**(5)** 300	301	300
Rx.free(150)		151	151		300	**(6)** 300
Rx.alloc	150	151	150	**(7)** 151	300	151

Table 1: *Illustrating why Rx-Tx interferences cause linearity, following the baseline allocation algorithm detailed in Figure 2. (Assume that all addresses are initially allocated.)*

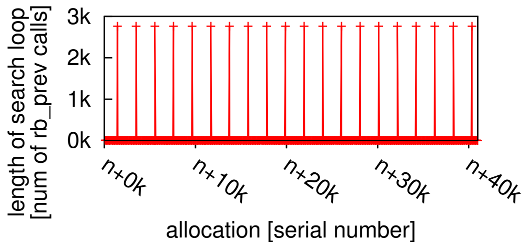


Figure 3: *The length of each alloc_iova search loop in a 40K (sub)sequence of alloc_iova calls performed by one Netperf run. One Rx-Tx interference leads to regular linearity.*

to iterate through the tree from H **(6)** to L **(7)**, inducing a linear overhead. Notably, once H is mapped for Rx, the pathology is repeated every time H is (de)allocated. This repetitiveness is experimentally demonstrated in Figure 3, showing the per-allocation number of rb_prev invocations. The calls are invoked in the loop in alloc_iova while searching for a free IOVA.

We show below that the implications of long-lasting ring interference can be dreadful in terms of performance. How, then, is it possible that such a deficiency is overlooked? We contend that the reason is twofold. The first is that commodity I/O devices were slow enough in the past such that IOVA allocation linearity did not matter. The second reason is the fact that using the IOMMU hardware is slow and incurs a high price, motivating the deferred protection safety/performance tradeoff. Being that slow, the hardware served as a scapegoat, wrongfully held accountable for most of the overhead penalty and masking the fact that software is equally to blame.

6 The EIOVAR Optimization

Suffering from frequent linear allocations, the baseline IOVA allocator is ill-suited for high-throughput I/O devices that are capable of performing millions of I/O transactions per second. It is too slow. One could proclaim that this is just another case of a special-purpose allocator proved inferior to a general-purpose allocator and argue that the latter should be favored over the former despite the notable differences between the two as listed in §4. We contend, however, that the simple, repetitive, and inherently ring-induced nature of the workload can be adequately served by the existing simplistic allocator—with only a small, minor change—such that the modified version is able to consistently support fast (de)allocations.

We propose the EIOVAR optimization (Efficient IOVA allocatoR), which rests of the following observation. I/O devices that stress the intra-OS protection mapping layer are not like processes, in that the size of their virtual address spaces is relatively small, inherently bounded by the size of their rings. A typical ring size n is a few hundreds or a few thousands of entries. The number of per-device virtual page addresses that the IOVA allocator must simultaneously support is proportional to the ring size, which means it is likewise bounded and relatively small. Moreover, unlike "regular" memory allocators, the IOVA mapping layer does not allocate real memory pages. Rather, it allocates integer identifiers for those pages. Thus, it is reasonable to keep O(n) of these identifiers alive under the hood for quick (de)allocation, without really (de)allocating them (in the traditional, malloc sense of (de)allocation).

In numerous experiments with multiple devices and workloads, the maximal number of per-device different IOVAs we have observed is 12K. More relevant is that, across all experiments, the maximal number of previously-allocated-but-now-free IOVAs has never exceeded 668 (and was 155 on average). Additionally, as noted earlier, the allocated IOVA ranges have a power of two size $H - L + 1 = 2^j$, where j is overwhelmingly 0. EIOVAR leverages these workload characteristic to efficiently cache freed IOVAs so as to satisfy future allocations quickly, similarly to what regular memory allocators do when allocating real memory [13, 14, 15, 29, 40, 53, 55].

EIOVAR is a thin layer that masks the red-black tree, resorting to using it only when EIOVAR cannot fulfill IOVA allocation on its own using previously freed elements. When configured to have enough capacity, all tree allocations that EIOVAR is unable to mask are assured to be fast and occur in constant time.

EIOVAR's main data structure is a one-dimensional array called "the freelist", or f for short. The array consists of M linked lists of IOVA ranges. Lists are empty upon initialization. When an IOVA range $[L,H]$ whose size is $H - L + 1 = 2^j$ is freed, instead of actually freeing it, EIOVAR adds it to the head of the linked list of the corresponding exponent, namely, to $f[j]$. Because most ranges are comprised of one page ($H = L$), most ranges end up in the $f[0]$ list after they are freed. The upper bound on the size of the ranges supported by EIOVAR is 2^{M+12} bytes (assuming 2^{12} = 4KB pages), as EIOVAR allocates page numbers. Thus, $M = 28$ is enough, allowing for up to a terabyte range.

EIOVAR allocation performs the reverse operation of freeing. When a range whose exponent is j is being allocated, EIOVAR removes the head of the $f[j]$ linked list in order to satisfy the allocation request. EIOVAR resorts to utilizing the baseline red-black tree only if a suitable range is not found in the freelist.

When no limit is imposed on the freelist, after a very short while, all EIOVAR (de)allocation operations are satisfied by f due to the inherently limited size of the ring-induced workload. All freelist (de)allocations are performed in constant time, taking 50-150 cycles per operation. Initial allocations that EIOVAR satisfies by resorting to the baseline tree are likewise done in constant time, because the freelist is limitless and so the tree never observes deallocations, which means its cache node C always points to its smallest, leftmost node (Figure 2).

We would like to make sure that the freelist is compact and is not effectively leaking memory. To bound the size of the freelist, EIOVAR has a parameter k that serves as f's maximal capacity of freed IOVAs. We use the EIOVAR$_k$ notation to express this limit, with $k = \infty$ indicating no upper bound. We demonstrate that setting k to be a relatively small number is equivalent to setting it to ∞, because the number of previously-allocated-but-now-free IOVAs is constrained by the size of the corresponding ring. Consequently, we can be certain that the freelist of EIOVAR$_\infty$ is a compact. At the same time, $k = \infty$ guarantees that (de)allocations are always satisfied in constant time.

6.1 EIOVAR with Strict Protection

To understand the behavior and effect of EIOVAR, we begin by analyzing five EIOVAR$_k$ variants as compared to the baseline under strict protection, where IOVAs are (de)allocated immediately before and after the associated DMAs. We use the standard Netperf stream benchmark that maximizes throughput on one TCP connection. We initially restart the NIC interface for each allocation variant (thus clearing IOVA structures), and then we execute the benchmark iteratively. The exact experimental setup is described in §7. The results are shown in Figure 4.

Figure 4a shows that the throughput of all EIOVAR variants is similar and is 20%–60% better than the baseline. The baseline gradually decreases except for the last iteration. Figure 4b highlights why even EIOVAR$_1$ is sufficient to provide the observed benefit. It plots the rate of IOVA allocations that are satisfied by the freelist, showing that $k = 1$ is enough to satisfy nearly all allocations. This result indicates that each call to free_iova is followed by alloc_iova, such that the IOVA freed by the former is returned by the latter, coinciding with the ideal scenario outlined in §4. Figure 4c supports this observation by depicting the average size of the freelist. The average of EIOVAR$_1$ is inevitably 0.5, as every allocation and deallocation contributes to the average 1 and 0 respectively. Larger k values are similar, with an average of 2.5 be-

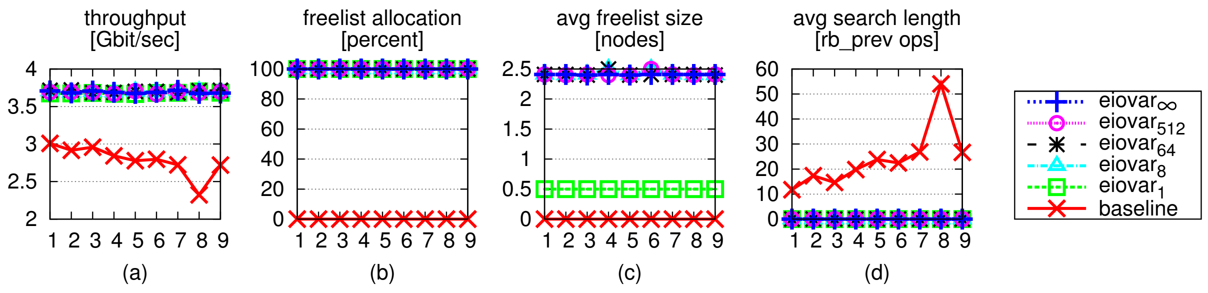


Figure 4: *Netperf TCP stream iteratively executed under strict protection. The x axis shows the iteration number.*

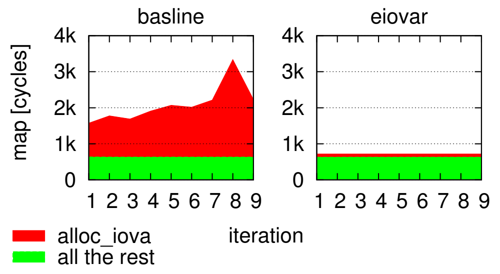


Figure 5: *Cycles breakdown of map with Netperf/strict.*

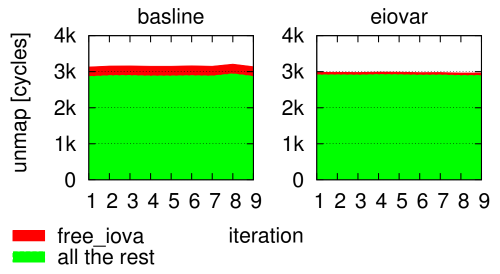


Figure 6: *Cycles breakdown of unmap with Netperf/strict.*

cause of two additional (de)allocation that are performed when Netperf starts running and that remain in the freelist thereafter. Figure 4d shows the average length of the 'while' loop from Figure 2, which searches for the next free IOVA. It depicts a rough mirror image of Figure 4a, indicating throughput is tightly negatively correlated with the traversal length.

Figure 5 (left) shows the time it takes the baseline to map an IOVA, separating allocation from the other activities. Whereas the latter remains constant, the former exhibits a trend identical to Figure 4d. Conversely, the alloc_iova time of EIOVAR (Figure 5, right) is negligible across the board. EIOVAR is immune to long-lasting ring interface, as interfering transactions are absorbed by the freelist and reused in constant time.

6.2 EIOVAR with Deferred Protection

Figure 6 is similar to Figure 5, but it pertains to the unmap operation rather than to map. It shows that the duration of free_iova remains stable across iterations with both EIOVAR and the baseline. EIOVAR deallocation is still faster as it is performed in constant time whereas the baseline is logarithmic. But most of the overhead is not due to free_iova. Rather, it is due to the costly invalidation that purges the IOVA from the IOTLB to protect the corresponding target buffer. This is the aforementioned hardware overhead that motivated deferred protection, which amortizes the cost by delaying invalidations until enough IOVAs are accumulated and then processing all of them together. As noted, deferring the invalidations trades off safety for performance, because the relevant memory is accessible by the device even though it is already used by the kernel for other purposes.

Figure 7 compares between the baseline and the EIOVAR variants under deferred protection. Interestingly, the resulting throughput divides the variants into two, with EIOVAR_{512} and EIOVAR_{∞} above 6Gbps and all the rest at around 4Gbps (Figure 7a). We again observe a strong negative correlation between the throughput and the length of the search to find the next free IOVA (Figure 7a vs. 7d).

In contrast to the strict setup (Figure 4), here we see that EIOVAR variants with smaller k values roughly perform as bad as the baseline. This finding is somewhat surprising, because, e.g., 25% of the allocations of EIOVAR_{64} are satisfied by the freelist (Figure 7b), which should presumably improve its performance over the baseline. A finding that helps explain this result is noticing that the average size of the EIOVAR_{64} freelist is 32 (Figure 7c), even though it is allowed to hold up to $k = 64$ elements. Notice that EIOVAR_{∞} holds around 128 elements on average, so we know there are enough deallocations to fully populate the EIOVAR_{64} freelist. One might therefore expect that the latter would be fully utilized, but it is not.

The average size of the EIOVAR_{64} freelist is 50% of its capacity due to the following reason. Deferred invalidations are aggregated until a high-water mark W (kernel parameter) is reached, and then all the W addresses are deallocated in bulk.[2] When $k < W$, the freelist fills up to

[2]They cannot be freed before they are purged from the IOTLB, or else they could be re-allocated, which would be a bug since their stale mappings might reside in the IOTLB and point to somewhere else.

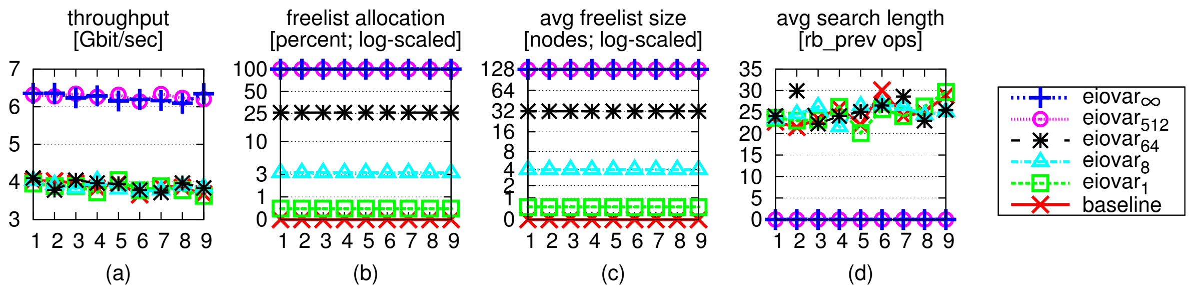


Figure 7: *Netperf TCP stream iteratively executed under deferred protection. The x axis shows the iteration number.*

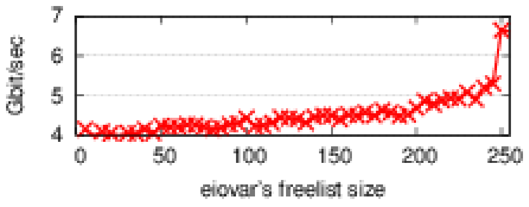


Figure 8: *Under deferred protection,* EIOVAR$_k$ *eliminates costly linear searches when k exceeds the high-water mark W.*

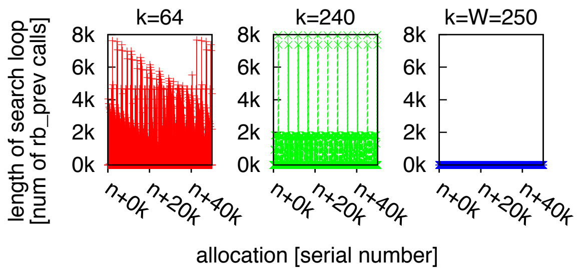


Figure 9: *Length of the* alloc_iova *search loop under the* EIOVAR$_k$ *deferred protection regime for three k values when running Netperf TCP Stream. Bigger capacity implies that the searches become shorter on average. Big enough capacity ($k \geq W = 250$) eliminates the searches altogether.*

hold k elements, which become $k-1$ after the subsequent allocation, and then $k-2$ after the next allocation, and so on until zero is reached, yielding an average size of $\frac{1}{k+1}\Sigma_{j=0}^{k} j \approx k/2$ as our measurements show.

Importantly, when $k < W$, EIOVAR$_k$ is unable to absorb all the W consecutive deallocations. The remaining $W-k$ deallocations are thus freed by the baseline free_iova. Thus, only k of the W subsequent allocation are satisfied by the freelist, and the remaining $W-k$ are serviced by the baseline alloc_iova. The baseline free_iova and alloc_iova are therefore regularly invoked in an uncoordinated way despite the freelist. As described in §5, the interplay between these two routines eventually causes long-lasting ring interference that induces repeated linear searches. In contrast, when k is big enough ($\geq W$), the freelist has sufficient capacity to absorb all W deallocations, which are then used to satisfy the subsequent W allocations and thus secure the conditions for preventing the harmful effect.

Figure 8 demonstrates this threshold behavior, depicting the throughput as a function of the maximal freelist size k. Increasingly bigger k slowly improves performance, as more—but not yet all—allocations are served by the freelist. When k reaches $W = 250$, the freelist is finally big enough, and the throughput suddenly increases by 26%. Figure 9 provides further insight into this result. It shows the per-allocation length of the loop within alloc_iova that iterates through the red-black tree in search for the next free IOVA (similarly to Figure 3). The subgraphs correspond to 3 points from Figure 8 with k values 64, 240, and 250. We see that the smaller k (left) yields longer searches relative to the bigger k (middle), and that the length of the search becomes zero when $k = W$ (right).

7 Evaluation

Experimental Setup We implement EIOVAR in the Linux kernel, and we experimentally compare its performance against the baseline IOVA allocation. In an effort to attain more general results, we conducted the evaluation using two setups involving two different NICs with two corresponding different device drivers that generate different workloads for the IOVA allocation layer.

The *Mellanox setup* consists of two identical Dell PowerEdge R210 II Rack Servers that communicate through Mellanox ConnectX3 40Gbps NICs. The NICs are connected back to back configured to use Ethernet. One machine is the server and the other is a workload generator client. Each machine has 8GB 1333MHz memory and a single-socket 4-core Intel Xeon E3-1220 CPU running at 3.10GHz. The chipset is Intel C202, which supports VT-d, Intel's Virtualization Technology that provides IOMMU functionality. We configure the server to utilize one core only, and we turn off all power optimizations—sleep states (C-states) and dynamic voltage and frequency scaling (DVFS)—to avoid reporting artifacts caused by nondeterministic events. The two machines run Ubuntu 12.04 and utilize the Linux 3.4.64 kernel.

The *Broadcom setup* is similar, with the difference that: the two R210 machines communicate through Broadcom NetXtreme II BCM57810 10GbE NICs (connected via a

CAT7 10GBASE-T cable for fast Ethernet); have 16GB memory; and run the Linux 3.11.0 kernel.

The drivers of the Mellanox and Broadcom NICs differ in many respects. Notably, the Mellanox driver uses more ring buffers and allocates more IOVAs (we observed around 12K addresses for Mellanox and 3K for Broadcom). In particular, the Mellanox driver uses two buffers per packet and hence two IOVAs, whereas the Broadcom driver allocates only one buffer and thus only one IOVA.

Benchmarks We use the following benchmarks to drive our experiments. Netperf TCP stream [37] is a standard tool to measure networking throughput. It attempts to maximize the amount of data sent over one TCP connection, simulating an I/O-intensive workload. This is the benchmark used when studying long-lasting ring interference (§5) and the impact of k on EIOVAR$_k$ (§6). We use the default 16KB message size unless otherwise stated.

Netperf UDP RR (request-response) is the second canonical configuration of Netperf. It models a latency sensitive workload by repeatedly sending a single byte and waiting for a matching single byte response. The latency is then calculated as the inverse of the observed number of transactions per second.

Apache [23, 24] is a HTTP web server. We drive it with ApacheBench [5] (a.k.a. "ab"), a workload generator distributed with Apache. ApacheBench assess the number of concurrent requests per second that the server is capable of handling by requesting a static page of a given size from within several concurrent threads. We run it on the client machine configured to generate 100 concurrent requests. We use two instances of the benchmark to request a smaller (1KB) and a bigger (1MB) file. Logging is disabled to avoid disk write overheads.

Memcached [25] is an in-memory key-value storage server. It is used, e.g., by websites for caching results of slow database queries, thus improving the sites' overall performance. We run Memslap [1] (part of the libmemcached client library) on the client machine, generating requests and measuring the completion rate. By default, Memslap generates a random workload comprised of 90% get and 10% set operations. Unless otherwise stated, Memslap is set to use 16 concurrent requests.

Methodology Before running each benchmark, we shut down and bring up the interface of the NIC using the ifconfig utility, such that the IOVA allocation is redone from scratch using a clean tree, clearing the impact of previous harmful long-lasting ring interferences. We then iteratively run the benchmark 150 times, such that individual runs are configured to take about 20 seconds. We present the corresponding results, on average.

Results Figure 10 shows the resulting average performance for the Mellanox (top) and Broadcom (bottom) setups. Higher numbers indicate better throughput in all cases but for Netperf RR, which depicts latency (inverse of throughput). The corresponding normalized values—specifying relative improvement—are shown in the first part of Table 2. Here, for consistency, the normalized throughput is shown for all benchmarks including RR.

Mellanox Setup We first examine the results of the Mellanox setup (left of Table 2). In the topmost part, we see that EIOVAR yields throughput 1.07–4.58x better than the baseline, and that improvements are more pronounced under strict protection. The second part of the table shows that the improved performance of EIOVAR is due to reducing the average IOVA allocation time by 1–2 orders of magnitude, from up to 50K cycles to around 100–200. EIOVAR further reduces the average IOVA deallocation time by about 75%–85%, from around 250–550 cycles to around 65–85 (4th part of the table).

As expected, the duration of the IOVA allocation routine is tightly correlated to the length of the search loop within this routine, such that a longer loop implies a longer duration (3rd part of Table 2). Notice, however, that there is not necessarily such a direct correspondence between EIOVAR's throughput improvement (1st part of table) and the associated IOVA allocation overhead (2nd part). The reason: latency sensitive applications are less affected by the allocation overhead, because other components in their I/O paths have higher relative weights. For example, under strict protection, the latency sensitive Netperf RR has higher allocation overhead as compared to the throughput sensitive Netperf Stream (10,269 cycles vs. 7,656, respectively), yet the throughput improvement of RR is smaller (1.27x vs. 2.37x). Similarly, the IOVA allocation overhead of Apache/1KB is higher than that of Apache/1MB (49,981 cycles vs. 17,776), yet its throughput improvement is lower (2.35x vs. 3.65x).

While there is not necessarily a direct connection between throughput and allocation overheads when examining strict safety only, the connection becomes apparent when comparing strict to deferred protection. Clearly, the benefit of EIOVAR in terms of throughput is greater under strict protection because the associated baseline allocation overheads are higher than that of deferred protection (7K–50K cycles for strict vs. 2K–3K for deferred).

Broadcom Setup Let us now examine the results of the Broadcom setup (right of Table 2). Strict EIOVAR yields throughput that is 1.07–2.35x better than the baseline. Deferred EIOVAR, on the other hand, only improves the throughput by up to 10%, and, in the case of Netperf Stream and Apache/1MB, it offers no improvement. Thus, while still significant, throughput improvements in this setup are less pronounced. The reason for this difference is twofold. First, as noted above, the driver of the Mellanox NIC utilizes more rings and more IOVAs, increasing the load on the IOVA allocation layer relative to the Broad-

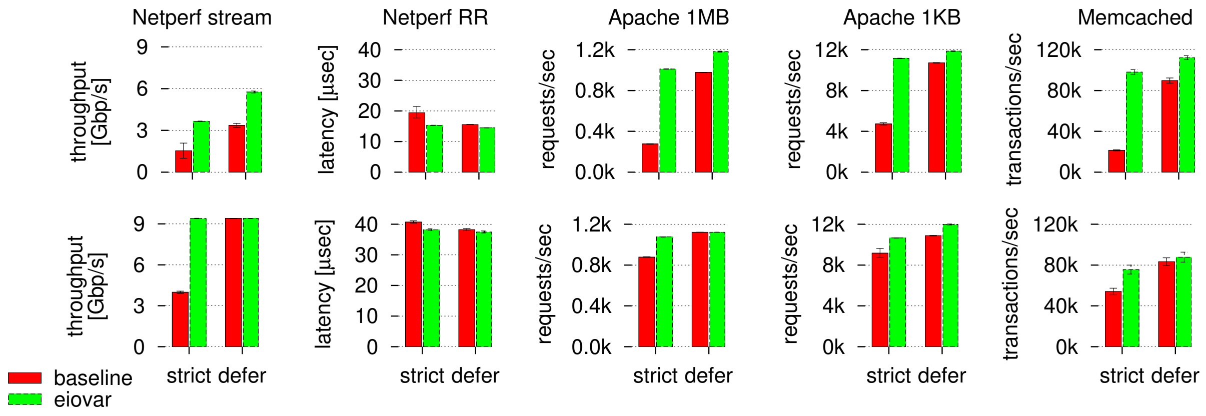


Figure 10: *The performance of baseline vs.* EIOVAR *allocation, under strict and deferred protection regimes for the Mellanox (top) and Broadcom (bottom) setups. Except for in the case of Netperf RR, higher values indicated better performance. Error bars depict the standard deviation (sometimes too small to be seen).*

com driver and generating more opportunities for ring interference. This difference is evident when comparing the duration of alloc_iova in the two setups, which is significantly lower in the Broadcom case. In particular, the average allocation time in the Mellanox setup across all benchmarks and protection regimes is about 15K cycles, whereas it is only about 3K cycles in the Broadcom setup.

The second reason for the less pronounced improvements in the Broadcom setup is that the Broadcom NIC imposes a 10 Gbps upper bound on the bandwidth, which is reached in some of the benchmarks. Specifically, the aforementioned Netperf Stream and Apache/1MB—which exhibit no throughput improvement under deferred EIOVAR—hit this limit. These benchmarks are already capable of obtaining line rate (maximal throughput) in the baseline/deferred configuration, so the lack of throughput improvement in their case should come as no surprise. Importantly, when evaluating I/O performance in a setting whereby the I/O channel is saturated, the interesting evaluation metric ceases to be throughput and becomes CPU usage. Namely, the question becomes which system is capable of achieving line rate using fewer CPU cycles. The bottom/right part of Table 2 shows that EIOVAR is indeed the more performant alternative, using 21% less CPU cycles in the case of the said Netperf Stream and Apache/1MB under deferred protection. (In the Mellanox setup, it is the CPU which is saturated in all cases but the latency sensitive Netperf RR.)

Deferred Baseline vs. Strict EIOVAR We explained above that deferred protection trades off safety to get better performance. We now note that, by Figure 10, the performance attained by EIOVAR when strict protection is employed is similar to the performance of the baseline configuration that uses deferred protection (the default in Linux). Specifically, in the Mellanox setup, on average, strict EIOVAR achieves 5% higher throughput than the deferred baseline, and in the Broadcom setup EIOVAR achieves 3% lower throughput. Namely, if strict EIOVAR is made the default, it will simultaneously deliver similar performance *and* better protection as compared to the current default configuration.

Different Message Sizes The default configuration of Netperf Stream utilizes a 16KB message size, which is big enough to optimize throughput. Our next experiment systematically explores the performance tradeoffs when utilizing smaller message sizes. Such messages can overwhelm the CPU and thus reduce the throughput. Another issue that might negatively affect the throughput of small packets is the maximal *number* of packets per second (PPS), which NICs commonly impose in conjunction with an upper bound on the throughput. (For example, the specification of our Broadcom NIC lists a maximal rate of 5.7 million PPS [33], and a rigorous experimental evaluation of this NIC reports that a single port in it is capable of delivering less than half that much [21].)

Figure 11 shows the throughput (top) and consumed CPU (bottom) as a function of message size for strict (left) and deferred safety (right) using the Netperf Stream benchmark in the Broadcom setup. With a 64B message size, the PPS limit dominates the throughput in all four configurations. Strict/baseline saturates the CPU with a message size as small as 256B; from that point on it achieves the same throughput (4Gbps), because the CPU remains its bottleneck. The other three configurations enjoy a gradually increasing throughput until line rate is reached. However, to achieve the same level of throughput, strict/EIOVAR requires more CPU than deferred/baseline, which in turn requires more CPU than deferred/EIOVAR.

Concurrency We next experiment concurrent I/O streams, as concurrency amplifies the harmful long-lasting ring interference. Figure 12 depicts the results of running

Mellanox	*protect*	*benchmark*	*baseline*	*EiovaR*	*diff*
throughput (normalized)	strict	Netperf stream	1.00	2.37	+137%
		Netperf RR	1.00	1.27	+27%
		Apache 1MB	1.00	3.65	+265%
		Apache 1KB	1.00	2.35	+135%
		Memcached	1.00	4.58	+358%
	defer	Netperf stream	1.00	1.71	+71%
		Netperf RR	1.00	1.07	+7%
		Apache 1MB	1.00	1.21	+21%
		Apache 1KB	1.00	1.11	+11%
		Memcached	1.00	1.25	+25%
alloc (cycles)	strict	Netperf stream	7656	88	-99%
		Netperf RR	10269	175	-98%
		Apache 1MB	17776	128	-99%
		Apache 1KB	49981	204	-100%
		Memcached	50606	151	-100%
	defer	Netperf stream	2202	103	-95%
		Netperf RR	2360	183	-92%
		Apache 1MB	2085	130	-94%
		Apache 1KB	2642	206	-92%
		Memcached	3040	171	-94%
search (length)	strict	Netperf stream	153	0	-100%
		Netperf RR	206	0	-100%
		Apache 1MB	381	0	-100%
		Apache 1KB	1078	0	-100%
		Memcached	893	0	-100%
	defer	Netperf stream	32	0	-100%
		Netperf RR	32	0	-100%
		Apache 1MB	30	0	-100%
		Apache 1KB	33	0	-100%
		Memcached	33	0	-100%
dealloc / free (cycles)	strict	Netperf stream	289	66	-77%
		Netperf RR	446	87	-81%
		Apache 1MB	360	70	-81%
		Apache 1KB	565	85	-85%
		Memcached	525	73	-86%
	defer	Netperf stream	273	65	-76%
		Netperf RR	242	66	-73%
		Apache 1MB	278	65	-76%
		Apache 1KB	300	66	-78%
		Memcached	334	65	-80%
cpu (%)	strict	Netperf stream	100	100	+0%
		Netperf RR	32	29	-8%
		Apache 1MB	100	99	-0%
		Apache 1KB	99	98	-1%
		Memcached	100	100	+0%
	defer	Netperf stream	100	100	+0%
		Netperf RR	30	29	-5%
		Apache 1MB	99	99	-0%
		Apache 1KB	98	98	-0%
		Memcached	100	100	+0%

Broadcom	*protect*	*benchmark*	*baseline*	*EiovaR*	*diff*
throughput (normalized)	strict	Netperf stream	1.00	2.35	+135%
		Netperf RR	1.00	1.07	+7%
		Apache 1MB	1.00	1.22	+22%
		Apache 1KB	1.00	1.16	+16%
		Memcached	1.00	1.40	+40%
	defer	Netperf stream	1.00	1.00	+0%
		Netperf RR	1.00	1.02	+2%
		Apache 1MB	1.00	1.00	+0%
		Apache 1KB	1.00	1.10	+10%
		Memcached	1.00	1.05	+5%
alloc (cycles)	strict	Netperf stream	14878	70	-100%
		Netperf RR	3359	100	-97%
		Apache 1MB	1469	74	-95%
		Apache 1KB	2527	116	-95%
		Memcached	5797	110	-98%
	defer	Netperf stream	1108	96	-91%
		Netperf RR	1029	118	-89%
		Apache 1MB	833	88	-89%
		Apache 1KB	1104	133	-88%
		Memcached	1021	130	-87%
search (length)	strict	Netperf stream	345	0	-100%
		Netperf RR	68	0	-100%
		Apache 1MB	27	0	-100%
		Apache 1KB	39	0	-100%
		Memcached	128	0	-100%
	defer	Netperf stream	13	0	-100%
		Netperf RR	9	0	-100%
		Apache 1MB	9	0	-100%
		Apache 1KB	9	0	-100%
		Memcached	9	0	-100%
dealloc / free (cycles)	strict	Netperf stream	294	47	-84%
		Netperf RR	282	48	-83%
		Apache 1MB	250	50	-80%
		Apache 1KB	425	52	-88%
		Memcached	342	47	-86%
	defer	Netperf stream	268	47	-82%
		Netperf RR	273	47	-83%
		Apache 1MB	234	47	-80%
		Apache 1KB	279	47	-83%
		Memcached	276	47	-83%
cpu (%)	strict	Netperf stream	100	53	-49%
		Netperf RR	13	12	-12%
		Apache 1MB	99	99	-0%
		Apache 1KB	98	98	-0%
		Memcached	99	95	-4%
	defer	Netperf stream	55	44	-21%
		Netperf RR	12	11	-7%
		Apache 1MB	91	72	-21%
		Apache 1KB	98	98	-0%
		Memcached	93	92	-2%

Table 2: *Summary of the results obtained with the Mellanox setup (left) and the Broadcom setup (right).*

Memcached in the Mellanox setup with an increasing number of clients. The left sub-graph reveals that the baseline allocation hampers scalability, whereas EIOVAR allows the benchmark to scale such that it is up to 5.5x more performant than the baseline (with 32 clients). The right sub-graphs highlights why, showing that the baseline IOVA allocation becomes costlier proportionally to the number of clients, whereas EIOVAR allocation remains negligible across the board.

FreeBSD We hypothesize that, like Linux, other OSes drive the IOMMU with suboptimal software, likely due to the perception that the IOMMU hardware is slow, possibly combined with the fact that I/O devices that are fast enough to significantly suffer from the consequences have become prevalent fairly recently. We test this hypothesis by studying the IOMMU mapping layer of FreeBSD. Our hypothesis coincides with the announcement of IOMMU support being added to FreeBSD, which says that "it

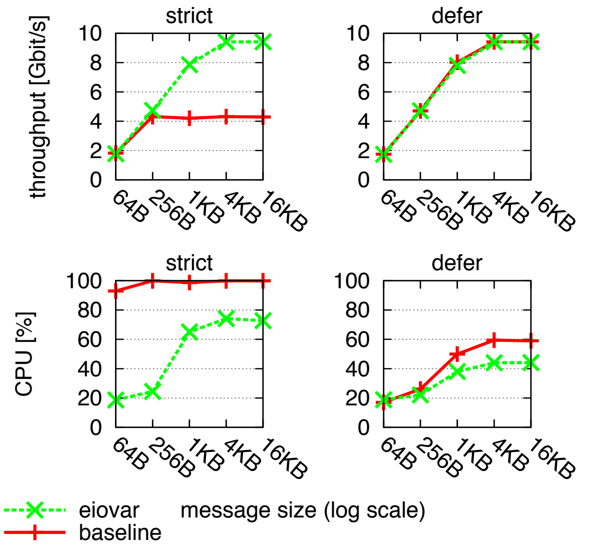


Figure 11: *Netperf Stream throughput (top) and used CPU (bottom) for different message sizes in the Broadcom setup.*

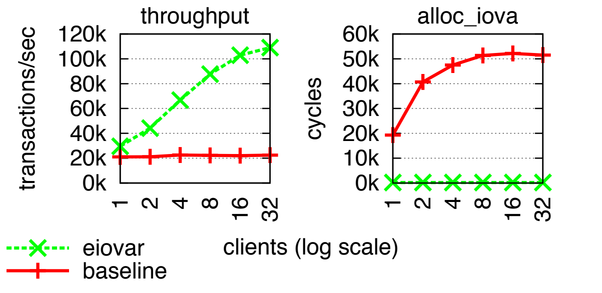


Figure 12: *Impact of increased concurrency on Memcached in the Mellanox setup.* EIOVAR *allows the performance to scale.*

is known that IOMMU adds overhead due to the mapping and unmapping for each I/O [and therefore it is] not plan[ned] to enable [the] IOMMU by default" [10].

We find that, similarly to Linux, FreeBSD uses a red-black tree for IOVA space management. Although it does not employ the problematic cached node optimization, the relevant source code can call fall back to a linear iteration through the tree nodes upon allocation. The comment preceding the linear iteration acknowledges that "this falls back to linear iteration over the free space in the high region"; however, the comment further notes that the said "high regions are almost unused" [26].

Using DTrace, the dynamic tracing tool [28], we profiled the IOVA mapping layer of FreeBSD while running the Netperf TCP stream benchmark. We measured each function along the call stack in a separate run, because multiple probe points affected the perceived results. Table 3 show the outcome, indicating that the FreeBSD IOMMU mapping layer overheads are larger than those of baseline Linux (compare with left of Figures 5–6). Specifically, whereas FreeBSD IOVA allocation is comparable

	map	*unmap*
iova	1,103	2,178
all the rest	8,557	13,825
total	9,660	16,003

Table 3: *FreeBSD IOMMU mapping layer overheads in cycles. (Compare with Linux's overheads in Figures 5–6.)*

to that of Linux, IOVA freeing takes an order of magnitude longer, and the (un)mapping is 4–5x slower altogether.

Our profiling reveals some of the root causes for these overheads. The aforementioned linear iteration remained inactive, as promised. But IOVA allocation turned out to nevertheless require the traversal of 11 red-black tree nodes on average. And the tree was rebalanced in almost every deallocation, introducing an overhead that is considerably higher than that of baseline Linux.

In addition to its inefficient IOVA (de)allocation, FreeBSD makes several suboptimal implementation choices that significantly slow down its mapping layer as compared to Linux. For instance, when a page within the IOMMU page table hierarchy is no longer in use, Linux usually does not reclaim it, rightfully assuming that it is likely to get reused soon. Conversely, FreeBSD does reclaim such pages, thereby reducing the memory footprint somewhat at the cost of increased CPU overheads.

The most wasteful unoptimized FreeBSD code we observed relates to synchronizing the I/O page table hierarchy between the IOMMU and the CPU. Upon every unmapping (ctx_unmap_buf_locked), FreeBSD flushes all the cachelines of the corresponding page table, although merely flushing the affected page table entries (PTEs) would have been enough. We applied the latter optimization to the FreeBSD unmap code and thus shortened it by $\sim$10K cycles on average, which improved the throughput of Netperf from 530 to 935 Mbps (1.76x higher).[3]

Consequently, in according to our hypothesis, we find that the FreeBSD mapping layer consists of suboptimal code that allows for easy optimizations that dramatically boost performance, possibly due to the perception that IOMMU hardware overheads are inherently high.

8 Related Work

Several studies recognized the poor performance associated with using the IOMMU [4, 12, 18, 44, 50, 57, 59]. Willmann et al. suggested to alleviate IOMMU overheads somewhat via "shared mappings", creating only one mapping for buffers that happen to reside on the same page instead of associating each of them with a different IOVA [57]. Amit et al. proposed to use "optimistic teardown",

[3]We confirmed this optimization with the relevant FreeBSD maintainer [11] and committed a patch that will be included in the next FreeBSD release [3].

whereby unmappings are delayed for a few milliseconds in the hope they will get immediately reused, creating a riskier policy than deferred protection that is more performant [4]. They also proposed to transparently offload the (un)mapping activity to computational cores different than the ones that perform the I/O. These approaches leave the original, unoptimized code intact and therefore EIOVAR is complementary to them.

Tomonori suggested to manage the IOVA space using bitmaps instead of trees, reporting an improvement in performance of 9% [50, 51]. Cascardo showed that performance is greatly improved if the driver of the I/O device can be modified to perform much fewer (un)mappings [18]. In a followup work, we proposed to redesign the IOMMU hardware to directly support the ring-induced workload and thus provide strict safety within 0.77–1.00x the throughput, 1.00–1.04x the latency, and 1.00–1.22x the CPU consumption of a system without an IOMMU [44].

Using freelists to speed up object allocation—as in EIOVAR—is a standard technique among memory allocators [13, 14, 15, 29, 40, 53, 55]. We discuss the contributes of this paper relative to such allocators in the next section.

9 Discussion, Conclusions, Future Work

Clements et al. made the case that implementations of OS kernels can be made scalable if they are designed beforehand such that their system calls commute, contending that "this rule aids developers in building more scalable software" [20]. Conversely, Linus Torvalds proclaimed that "Linux is evolution, not intelligent design" [22], likely more accurately reflecting the manner by which OSes are built, typically using the simplest implementation until experience proves that this is the wrong way to go.

When implementing new kernel functionality, a linear algorithm is often favored as being the simplest. For example, such was the case with the original linear Linux scheduler, which survived a decade [47]. And such is still the case with vmalloc, which is the internal Linux kernel function that is responsible for allocating virtually contiguous memory [52] (as opposed to kmalloc, which allocates *physically* contiguous memory). The pro of favoring linearity is simplicity. The con is that it might hinder performance when assumptions change.

The Intel/Linux IOVA allocation algorithm admittedly models the vmalloc algorithm [43]. From examining the source code, we see that both use a red-black tree for storing address ranges; both cache the location of the last freed range; and both use the cache as the starting point for subsequent allocations, traversing the tree elements in search for a large enough hole. We are not aware of workloads that utilize vmalloc whose performance noticeably degrades as a consequence. We demonstrate, however, that I/O intensive workloads suffer greatly form the linearity of IOVA allocation, which is induced by the "long-term ring interference" pathology that we characterize.

We conjecture that this deficiency exists because the IOMMU has been falsely perceived as the main responsible party for the significant overheads of intra-OS protection, and possibly because I/O devices fast enough to be noticeably affected have become widespread only in the last few years. We support our conjecture with experimental data from both Linux and FreeBSD.

We employ the compact EIOVAR optimization that proxies IOVA (de)allocations, resorting to the underlying red-black tree only if EIOVAR is unable to satisfy requests with its freelist. EIOVAR makes the baseline allocator orders of magnitude faster, improving the performance of common benchmarks by up to 5.5x.

Using freed object caches for fast allocation similarly to EIOVAR's freelist is not new. It is a standard technique employed by memory allocators [13, 14, 15, 29, 40, 53, 55]. Our contribution lies not in inventing the technique but rather: in (1) noticing it is applicable to, and substantially improves the performance of, the IOMMU mapping layer, which goes against the common wisdom that the slowness of this layer is due to the slowness of the hardware; in (2) carefully characterizing the workload experienced by the mapping layer; and in (3) finding that the workload characteristics allow for even the most basic/minimal freelist mechanism to deliver high performance, since (3.1) allocation requests exclusively consist of rounded up power-of-two areas that accelerate IOTLB invalidations without wasting real memory, and since (3.2) the freelist population size is inherently constrained by the relatively small size of the associated ring, so it can be used without worrying that the population of the previously-allocated-but-now-free IOVAs would explode.

EIOVAR eliminates one serious bottleneck of the IOMMU mapping layer. But we suspect that other bottlenecks exist, notably in relation to its locking regime, which affects subsystems different than the IOVA allocator and might hinder scalability. In the future, we therefore intend to study how the mapping layer scales as core-count increases. Another interesting question we intend to study is whether it is possible, and how hard is it, to exploit the window of vulnerability inherent to deferred protection as compared to strict protection.

Acknowledgments We thank the reviewers, our shepherd Carl Waldspurger, Muli Ben-Yehuda, and Dror Feitelson for providing valuable feedback. This research was funded in part by the Intel Collaborative Research Institute for Computational Intelligence (ICRI-CI) and by the Israeli Ministry of Science and Technology grant #3-9779. Mellanox graciously donated NICs.

References

[1] Brian Aker. Memslap - load testing and benchmarking a server. http://docs.libmemcached.org/bin/memslap.html. libmemcached 1.1.0 documentation.

[2] AMD Inc. AMD IOMMU architectural specification, rev 2.00. http://support.amd.com/TechDocs/48882.pdf, Mar 2011.

[3] Nadav Amit and Konstantin Belousov. svn commit: r277023 – head/sys/x86/iommu, FreeBSD kerenl commit. https://lists.freebsd.org/pipermail/svn-src-head/2015-January/066777.html, Jan 2015. (Accessed: Jan 2015).

[4] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, and Assaf Schuster. vIOMMU: efficient IOMMU emulation. In *USENIX Annual Technical Conference (ATC)*, pages 73–86, 2011.

[5] Apachebench. http://en.wikipedia.org/wiki/ApacheBench.

[6] Apple Inc. Thunderbolt device driver programming guide: Debugging VT-d I/O MMU virtualization. https://developer.apple.com/library/mac/documentation/HardwareDrivers/Conceptual/ThunderboltDevGuide/DebuggingThunderboltDrivers/DebuggingThunderboltDrivers.html, 2013. (Accessed: May 2014).

[7] ARM Holdings. ARM system memory management unit architecture specification — SMMU architecture version 2.0. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0062c/IHI0062C_system_mmu_architecture_specification.pdf, 2013.

[8] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *ACM Eurosys*, pages 73–85, 2006.

[9] Michael Becher, Maximillian Dornseif, and Christian N. Klein. FireWire: all your memory are belong to us. In *CanSecWest applied security conference*, 2005.

[10] Konstantin Belousov. FreeBSD x86 IOMMU support (DMAR). http://lists.freebsd.org/pipermail/freebsd-arch/2013-May/014368.html, May 2013. (Accessed: Jan 2015).

[11] Konstantin Belousov. FreeBSD VT-d IOMMU implementation. Private email communication, Jan 2015.

[12] Muli Ben-Yehuda, Jimi Xenidis, Michal Ostrowski, Karl Rister, Alexis Bruemmer, and Leendert van Doorn. The price of safety: Evaluating IOMMU performance. In *Ottawa Linux Symposium (OLS)*, pages 9–20, 2007.

[13] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 117–128, 2000.

[14] Jeff Bonwick. The Slab allocator: An object-caching kernel memory allocator. In *USENIX Summer Annual Technical Conf*, pages 87–98, 1994.

[15] Jeff Bonwick and Jonathan Adams. Magazines and Vmem: Extending the Slab allocator to many CPUs and arbitrary resources. In *USENIX Annual Technical Conference (ATC)*, pages 15–44, 2001.

[16] James E.J. Bottomley. Dynamic DMA mapping using the generic device. https://www.kernel.org/doc/Documentation/DMA-API.txt. Linux kernel documentation.

[17] Brian D. Carrier and Joe Grand. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation*, 1(1):50–60, Feb 2014.

[18] Thadeu Cascardo. DMA API performance and contention on IOMMU enabled environments. http://events.linuxfoundation.org/images/stories/slides/lfcs2013_cascardo.pdf, 2013.

[19] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 73–88, 2001.

[20] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–17, 2013.

[21] Demartek, LLC. QLogic FCoE/iSCSI and IP networking adapter evaluation (previously: Broadcom BCM957810 10Gb). http://www.demartek.com/Reports_Free/Demartek_QLogic_57810S_FCoE_iSCSI_Adapter_Evaluation_2014-05.pdf, May 2014. (Accessed: May 2014).

[22] Dror G. Feitelson. Perpetual development: A model of the Linux kernel life cycle. *Journal of Systems and Software*, 85(4):859–875, 2012.

[23] The Apache HTTP server project. http://httpd.apache.org.

[24] Roy T. Fielding and Gail Kaiser. The Apache HTTP server project. *IEEE Internet Computing*, 1(4):88–90, Jul 1997.

[25] Brad Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004(124), Aug 2004.

[26] FreeBSD Foundation. x86/iommu/intel_gas.c, source code file of FreeBSD 10.1.0. https://github.com/freebsd/freebsd/blob/release/10.1.0/sys/x86/iommu/intel_gas.c#L447. (Accessed: Jan 2015).

[27] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. ELI: Bare-metal performance for I/O

virtualization. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 411–422, 2012.

[28] Brendan Gregg and Jim Mauro. *DTrace Dynamic Tracing in Oracle, Solaris, Mac OS X, and FreeBSD*. Prentice Hall, 2011.

[29] Dirk Grunwald and Benjamin Zorn. CustoMalloc: Efficient synthesized memory allocators. *Software Practice and Experience (SPE)*, 23(8):851–869, 1993. Aug.

[30] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Failure resilience for device drivers. In *IEEE/IFIP Annual International Conference on Dependable Systems and Networks (DSN)*, pages 41–50, 2007.

[31] Brian Hill. Integrating an EDK custom peripheral with a LocalLink interface into Linux. Technical Report XAPP1129 (v1.0), XILINX, May 2009.

[32] The hoard memory allocator. `http://www.hoard.org/`. (Accessed: May 2014).

[33] HP Development Company. Family data sheet: Broadcom NetXtreme network adapters for HP ProLiant Gen8 servers. `http://www.broadcom.com/docs/features/netxtreme_ethernet_hp_datasheet.pdf`, Aug 2013. Rev. 2. (Accessed: May 2014).

[34] IBM Corporation. PowerLinux servers — 64-bit DMA concepts. `http://pic.dhe.ibm.com/infocenter/lnxinfo/v3r0m0/topic/liabm/liabmconcepts.htm`. (Accessed: May 2014).

[35] IBM Corporation. AIX kernel extensions and device support programming concepts. `https://publib.boulder.ibm.com/infocenter/aix/v7r1/topic/com.ibm.aix.kernelext/doc/kernextc/kernextc_pdf.pdf`, 2013. (Accssed: May 2014).

[36] Intel Corporation. Intel virtualization technology for directed I/O, architecture specification - architecture specification - rev. 2.2. `http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf`, Sep 2013.

[37] Rick A. Jones. A network performance benchmark (revision 2.0). Technical report, Hewlett Packard, 1995. `http://www.netperf.org/netperf/training/Netperf.html`.

[38] Asim Kadav, Matthew J. Renzelmann, and Michael M. Swift. Tolerating hardware device failures in software. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 59–72, 2009.

[39] Anil S. Keshavamurthy. [patch -mm][Intel-IOMMU] optimize sg map/unmap calls. Linux Kernel Mailing List `https://lkml.org/lkml/2007/8/1/402`, Aug 2007.

[40] Doug Lea. A memory allocator. `http://g.oswego.edu/dl/html/malloc.html`, 2000.

[41] Doug Lea. malloc.c. `ftp://g.oswego.edu/pub/misc/malloc.c`, Aug 2012. (Accessed: May 2014).

[42] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 17–30, 2004.

[43] Documentation/intel-iommu.txt, Linux 3.18 documentation file. `https://www.kernel.org/doc/Documentation/Intel-IOMMU.txt`. (Accessed: Jan 2015).

[44] Moshe Malka, Nadav Amit, Muli Ben-Yehuda, and Dan Tsafrir. rIOMMU: Efficient IOMMU for I/O devices that employ ring buffers. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015. To appear.

[45] Vinod Mamtani. DMA directions and Windows. `http://download.microsoft.com/download/a/f/d/afdfd50d-6eb9-425e-84e1-b4085a80e34e/sys-t304_wh07.pptx`, 2007. (Accessed: May 2014).

[46] David S. Miller, Richard Henderson, and Jakub Jelinek. Dynamic DMA mapping guide. `https://www.kernel.org/doc/Documentation/DMA-API-HOWTO.txt`. Linux kernel documentation.

[47] Ingo Monar. Goals, design and implementation of the new ultra-scalable O(1) scheduler. `http://lxr.free-electrons.com/source/Documentation/scheduler/sched-design.txt?v=2.6.25`, Apr 2002.

[48] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 335–350, 2007.

[49] Michael Swift, Brian Bershad, and Henry Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems (TOCS)*, 23(1):77–110, Feb 2005.

[50] Fujita Tomonori. DMA representations sg_table vs. sg_ring IOMMUs and LLD's restrictions. In *USENIX Linux Storage and Filesystem Workshop (LSF)*, 2008. `https://www.usenix.org/legacy/events/lsf08/tech/IO_tomonori.pdf`.

[51] Fujita Tomonori. Intel IOMMU (and IOMMU for virtualization) performances. `https://lkml.org/lkml/2008/6/4/250`, Jun 2008. (Accessed: Jan 2015).

[52] Linus Torvalds and others. mm/vmalloc.c, source code file of Linux 3.17. `http://lxr.free-electrons.com/source/mm/vmalloc.c?v=3.17`. (Accessed: Jan 2015).

[53] Jim Van Sciver. Zone garbage collection. In *USENIX Mach Workshop*, pages 1–16, 1990.

[54] Carl Waldspurger and Mendel Rosenblum. I/O virtualization. *Communications of the ACM (CACM)*, 55(1):66–73, Jan 2012.

[55] C. B. Weinstock and W. A. Wulf. Quick Fit: An efficient algorithm for heap storage allocation. *ACM SIGPLAN Notices*, 23(10):141–148, Oct 1988.

[56] Dan Williams, Patrick Reynolds, Kevin Walsh, Emin Gün Sirer, and Fred B. Schneider. Device driver safety through a reference validation mechanism. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 241–254, 2008.

[57] Paul Willmann, Scott Rixner, and Alan L. Cox. Protection strategies for direct access to virtualized I/O devices. In *USENIX Annual Technical Conference (ATC)*, pages 15–28, 2008.

[58] Rafal Wojtczuk. Subverting the Xen hypervisor. In *Black Hat*, 2008. `http://www.blackhat.com/presentations/bh-usa-08/Wojtczuk/BH_US_08_Wojtczuk_Subverting_the_Xen_Hypervisor.pdf`. (Accessed: May 2014).

[59] Ben-Ami Yassour, Muli Ben-Yehuda, and Orit Wasserman. On the DMA mapping problem in direct device assignment. In *ACM International Systems and Storage Conference (SYSTOR)*, pages 18:1–18:12, 2010.

FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs

Da Zheng, Disa Mhembere, Randal Burns
Department of Computer Science
Johns Hopkins University

Joshua Vogelstein
Institute for Computational Medicine
Department of Biomedical Engineering
Johns Hopkins University

Carey E. Priebe
Department of Applied Mathematics and Statistics
Johns Hopkins University

Alexander S. Szalay
Department of Physics and Astronomy
Johns Hopkins University

Abstract

Graph analysis performs many random reads and writes, thus, these workloads are typically performed in memory. Traditionally, analyzing large graphs requires a cluster of machines so the aggregate memory exceeds the graph size. We demonstrate that a multicore server can process graphs with billions of vertices and hundreds of billions of edges, utilizing commodity SSDs with minimal performance loss. We do so by implementing a graph-processing engine on top of a user-space SSD file system designed for high IOPS and extreme parallelism. Our semi-external memory graph engine called FlashGraph stores vertex state in memory and edge lists on SSDs. It hides latency by overlapping computation with I/O. To save I/O bandwidth, FlashGraph only accesses edge lists requested by applications from SSDs; to increase I/O throughput and reduce CPU overhead for I/O, it conservatively merges I/O requests. These designs maximize performance for applications with different I/O characteristics. FlashGraph exposes a general and flexible vertex-centric programming interface that can express a wide variety of graph algorithms and their optimizations. We demonstrate that FlashGraph in semi-external memory performs many algorithms with performance up to 80% of its in-memory implementation and significantly outperforms PowerGraph, a popular distributed in-memory graph engine.

1 Introduction

Large-scale graph analysis has emerged as a fundamental computing pattern in both academia and industry. This has resulted in specialized software ecosystems for scalable graph computing in the cloud with applications to web structure and social networking [10, 20], machine learning [18], and network analysis [22]. The graphs are massive: Facebook's social graph has billions of vertices and today's web graphs are much larger.

The workloads from graph analysis present great challenges to system designers. Algorithms that perform edge traversals on graphs induce many small, random I/Os, because edges encode non-local structure among vertices and many real-world graphs exhibit a power-law distribution on the degree of vertices. As a result, graphs cannot be clustered or partitioned effectively [17] to localize access. While good partitions may be important for performance [8], leading systems partition natural graphs randomly [11].

Graph processing engines have converged on a design that *(i)* stores graph partitions in the aggregate memory of a cluster, *(ii)* encodes algorithms as parallel programs against the vertices of the graph, and *(iii)* uses either distributed shared memory [18, 11] or message passing [20, 10, 24] to communicate between non-local vertices. Placing data in memory reduces access latency when compared to disk drives. Network performance, required for communication between graph partitions, emerges as the bottleneck and graph engines require fast networks to realize good performance.

Recent work has turned back to processing graphs from disk drives on a single machine [16, 23] to achieve scalability without excessive hardware. These engines are optimized for the sequential performance of magnetic disk drives; they eliminate random I/O by scanning the entire graph dataset. This strategy can be wasteful for algorithms that access only small fractions of data during each iteration. For example, breadth-first search, a building block for many graph applications, only processes vertices in a frontier. PageRank [7] starts processing all vertices in a graph, but as the algorithm progresses, it narrows to a small subset of active vertices. There is a huge performance gap between these systems and in-memory processing.

We present FlashGraph, a semi-external memory graph-processing engine that meets or exceeds the performance of in-memory engines and allows graph problems to scale to the capacity of semi-external memory.

Semi-external memory [2, 22] maintains algorithmic vertex state in RAM and edge lists on storage. The semi-external memory model avoids writing data to SSDs. Only using memory for vertices increases the scalability of graph engines in proportion to the ratio of edges to vertices in a graph, more than 35 times for our largest graph of Web page crawls. FlashGraph uses an array of solid-state drives (SSDs) to achieve high throughput and low latency to storage. Unlike magnetic disk-based engines, FlashGraph supports selective access to edge lists.

Although SSDs can deliver high IOPS, we overcome many technical challenges to construct a semi-external memory graph engine with performance comparable to an in-memory graph engine. The throughput of SSDs are an order of magnitude less than DRAM and the I/O latency is multiple orders of magnitude slower. Also, I/O performance is extremely non-uniform and needs to be localized. Finally, high-speed I/O consumes many CPU cycles, interfering with graph processing.

We build FlashGraph on top of a user-space SSD file system called SAFS [32] to overcome these technical challenges. The set-associative file system (SAFS) refactors I/O scheduling, data placement, and data caching for the extreme parallelism of modern NUMA multiprocessors. The lightweight SAFS cache enables FlashGraph to adapt to graph applications with different cache hit rates. We integrate FlashGraph with the asynchronous user-task I/O interface of SAFS to reduce the overhead of accessing data in the page cache and memory consumption, as well as overlapping computation with I/O.

FlashGraph issues I/O requests carefully to maximize the performance of graph algorithms with different I/O characteristics. It reduces I/O by only accessing edge lists requested by applications and using compact external-memory data structures. It reschedules I/O access on SSDs to increase the cache hits in the SAFS page cache. It conservatively merges I/O requests to increase I/O throughput and reduces CPU overhead by I/O.

Our results show that FlashGraph in semi-external memory achieves performance comparable to its in-memory version and Galois [21], a high-performance, in-memory graph engine with a low-level API, on a wide-variety of algorithms that generate diverse access patterns. FlashGraph in semi-external memory mode significantly outperforms PowerGraph, a popular distributed in-memory graph engine. We further demonstrate that FlashGraph can process massive natural graphs in a single machine with relatively small memory footprint; e.g., we perform breadth-first search on a graph of 3.4 billion vertices and 129 billion edges using only 22 GB of memory. Given the fast performance and small memory footprint, we conclude that FlashGraph offers unprecedented opportunities for users to perform massive graph analysis efficiently with commodity hardware.

2 Related Work

MapReduce [9] is a general large-scale data processing framework. PEGASUS [13] is a popular graph processing engine whose architecture is built on MapReduce. PEGASUS respects the nature of the MapReduce programming paradigm and expresses graph algorithms as a generalized form of sparse matrix-vector multiplication. This form of computation works relatively well for graph algorithms such as PageRank [7] and label propagation [33], but performs poorly for graph traversal algorithms.

Several other works [14, 19] perform graph analysis using linear algebra with sparse adjacency matrices and vertex-state vectors as data representations. In this abstraction, PageRank and label propagation are efficiently expressed as sparse-matrix, dense-vector multiplication, and breadth-first search as sparse-matrix, sparse-vector multiplication. These frameworks target mathematicians and those with the ability to formulate and express their problems in the form of linear algebra.

Pregel [20] is a distributed graph-processing framework that allows users to express graph algorithms in vertex-centric programs using bulk-synchronous processing (BSP). It abstracts away the complexity of programming in a distributed-memory environment and runs users' code in parallel on a cluster. Giraph [10] is an open-source implementation of Pregel.

Many distributed graph engines adopt the vertex-centric programming model and express different designs to improve performance. GraphLab [18] and PowerGraph [11] prefer shared-memory to message passing and provide asynchronous execution. FlashGraph supports both pulling data from SSDs and pushing data with message passing. FlashGraph does provide asynchronous execution of vertex programs to overlap computing with data access. Trinity [24] optimizes message passing by restricting vertex communication to a vertex and its direct neighbors.

Ligra [25] is a shared-memory graph processing framework and its programming interface is specifically optimized for graph traversal algorithms. It is not as general as other graph engines such as Pregel, GraphLab, PowerGraph, and FlashGraph. Furthermore, Ligra's maximum supported graph size is limited by the memory size of a single machine.

Galois [21] is a graph programming framework with a low-level abstraction to implement graph engines. The core of the Galois framework is its novel task scheduler. The dynamic task scheduling in Galois is orthogonal to FlashGraph's I/O optimizations and could be adopted.

GraphChi [16] and X-stream [23] are specifically designed for magnetic disks. They eliminate random data access from disks by scanning the entire graph dataset in each iteration. Like graph processing frameworks built on top of MapReduce, they work relatively well for graph algorithms that require computation on all vertices, but share the same limitations, i.e., suboptimal graph traversal algorithm performance.

TurboGraph [12] is an external-memory graph engine optimized for SSDs. Like FlashGraph, it reads vertices selectively and fully overlaps I/O and computation. TurboGraph targets graph algorithms expressed in sparse matrix vector multiplication, so it is difficult to implement graph applications such as triangle counting. It uses much larger I/O requests than FlashGraph to read vertices selectively due to its external-memory data representation. Furthermore, it targets graph analysis on a single SSD or a small SSD array and does not aim at performance comparable to in-memory graph engines.

Abello et al. [2] introduced the semi-external memory algorithmic framework for graphs. Pearce et al. [22] implemented several semi-external memory graph traversal algorithms for SSDs. FlashGraph adopts and advances several concepts introduced by these works.

3 Design

FlashGraph is a semi-external memory graph engine optimized for any fast I/O device such as Fusion I/O or arrays of solid-state drives (SSDs). It stores the edge lists of vertices on SSDs and maintains vertex state in memory. FlashGraph runs on top of the set-associative file system (SAFS) [32], a user-space filesystem designed to realize both high IOPS and lightweight caching for SSD arrays on non-uniform memory and I/O systems.

We design FlashGraph with two goals: to achieve performance comparable to in-memory graph engines while realizing the increased scalability of the semi-external memory execution model; to have a concise and flexible programming interface to express a wide variety of graph algorithms, as well as their optimizations.

To optimize performance, we design FlashGraph with the following principles:
Reduce I/O: Because SSDs are an order of magnitude slower than RAM, FlashGraph saturates the I/O channel in many graph applications. Reducing the amount of I/O for a given algorithm directly improves performance. FlashGraph *(i)* compacts data structures, *(ii)* maximizes cache hit rates and *(iii)* performs selective data access to edge lists.
Perform sequential I/O when possible: Even though SSDs provide high IOPS for random access, sequential I/O always outperforms random I/O and reduces the CPU overhead of I/O processing in the kernel.
Overlap I/O and Computation: To fully utilize multicore processors and SSDs for data-intensive workloads, one must initiate many parallel I/Os and process data when it is ready.
Minimize wearout: SSDs wear out after many writes, especially for consumer SSDs. Therefore, it is important to minimize writes to SSDs. This includes avoiding writing data to SSDs during the application execution and reducing the necessity of loading graph data to SSDs multiple times for the same graph.

In practice, selective data access and performing sequential I/O conflict. Selective data access prevents us from generating large sequential I/O, while using large sequential I/O may bring in unnecessary data from SSDs in many graph applications. For SSDs, FlashGraph places a higher priority in reducing the number of bytes read from SSDs than in performing sequential I/O because the random (4KB) I/O throughput of SSDs today is only two or three times less than their sequential I/O. In contrast, hard drives have random I/O throughput two orders of magnitude smaller than their sequential I/O. Therefore, other external-memory graph engines such as GraphChi and X-stream place a higher priority in performing large sequential I/O.

3.1 SAFS

SAFS [32] is a user-space filesystem for high-speed SSD arrays in a NUMA machine. It is implemented as a library and runs in the address space of its application. It is deployed on top of the Linux native filesystem.

SAFS reduces overhead in the Linux block subsystem, enabling maximal performance from an SSD array. It deploys dedicated per-SSD I/O threads to issue I/O requests with Linux AIO to reduce locking overhead in the Linux kernel; it refactors I/Os from applications and sends them to I/O threads with message passing. Furthermore, it has a scalable, lightweight page cache that organizes pages in a hashtable and places multiple pages in a hashtable slot [31]. This page cache reduces locking overhead and incurs little overhead when the cache hit rate is low; it increases application-perceived performance linearly along with the cache hit rate.

To better support FlashGraph, we add an asynchronous user-task I/O interface to SAFS. This I/O interface supports general-purpose computation in the page cache, avoiding the pitfalls of Linux asynchronous I/O. To achieve maximal performance, SSDs require many parallel I/O requests. This could be achieved with user-initiated asynchronous I/O. However, this asynchronous I/O requires the allocation of user-space buffers in advance and the copying of data into these buffers. This creates processing overhead from copying and further pollutes memory with empty buffers waiting to be filled. When an application issues a large number of parallel I/O requests, the empty buffers account for substantial memory consumption. In the SAFS user-task programming interface, an application associates a user-defined task

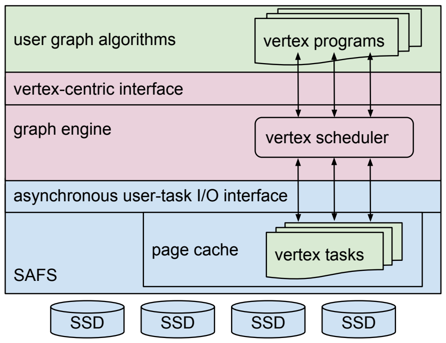


Figure 1: The architecture of FlashGraph.

with each I/O request. Upon completion of a request, the associated user task executes inside the filesystem, accessing data in the page cache directly. Therefore, there is no memory allocation and copy for asynchronous I/O.

3.2 The architecture of FlashGraph

We build FlashGraph on top of SAFS to fully utilize the high I/O throughput provided by the SSD array (Figure 1). FlashGraph solely uses the asynchronous user-task I/O interface of SAFS to reduce the overhead of accessing data in the page cache, memory consumption, as well as overlapping computation with I/O. FlashGraph uses the scalable, lightweight SAFS page cache to buffer the edge lists from SSDs so that FlashGraph can adapt to applications with different cache hit rates.

A graph algorithm in FlashGraph is composed of many vertex programs that run inside the graph engine. Each vertex program represents a vertex and has its own user-defined state and logic. The execution of vertex programs is subject to scheduling by FlashGraph. When vertex programs need to access data from SSDs, FlashGraph issues I/O requests to SAFS on behalf of the vertex programs and pushes part of their computation to SAFS.

3.3 Execution model

FlashGraph proceeds in iterations when executing graph algorithms, much like other engines. In each iteration, FlashGraph processes the vertices activated in the previous iteration. An algorithm ends when there are no active vertices in the next iteration.

As shown in Figure 2, FlashGraph splits a graph into multiple partitions and assigns a worker thread to each partition to process vertices. Each worker thread maintains a queue of active vertices within its own partition and executes user-defined vertex programs on them. FlashGraph's scheduler both manages the order of execution of active vertices and guarantees only a fixed number of running vertices in a thread.

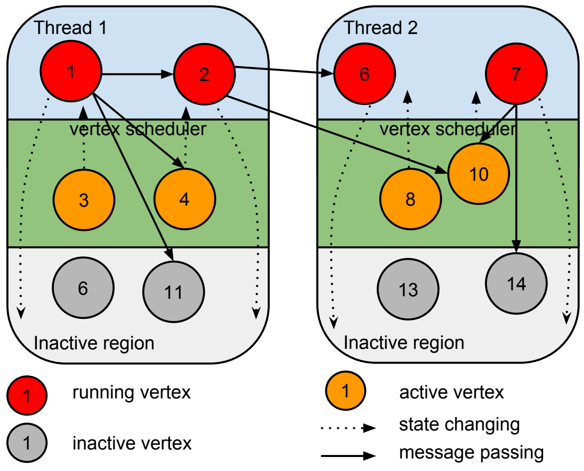


Figure 2: Execution model in FlashGraph.

There are three possible states for a vertex: *(i)* running, *(ii)* active, or *(iii)* inactive. A vertex can be activated either by other vertices or the graph engine itself. An active vertex enters the running state when it is scheduled. It remains in the running state until it finishes its task in the current iteration and becomes inactive. The running vertices interact with other vertices via message passing.

3.4 Programming model

FlashGraph aims at providing a flexible programming interface to express a variety of graph algorithms and their optimizations. FlashGraph adopts the vertex-centric programming model commonly used by other graph engines such as Pregel [20] and PowerGraph [11]. In this programming model, each vertex maintains vertex state and performs user-defined tasks based on its own state. A vertex affects the state of others by sending messages to them as well as activating them. We further allow a vertex to read the edge list of any vertex from SSDs.

The `run` method (Figure 3) is the entry point of a vertex program in an iteration. It is scheduled and executed exactly once on each active vertex. It is designed intentionally to have only access the vertex's own state in this method. A vertex must explicitly request its own edge list before accessing it because it is common that vertices are activated but do not perform any computation. Reading a vertex's edge list by default before executing its `run` method wastes I/O bandwidth.

The rest of FlashGraph's programming interface is event-driven to overlap computation and I/O, and receive notifications from the graph engine and other vertices. A vertex may receive three types of events:

- when it receives the edge list of a vertex, FlashGraph executes its `run_on_vertex` method.

```
class vertex {
  // entry point (runs in memory)
  void run(graph_engine &g);
  // per vertex computation (runs in the SAFS page cache)
  void run_on_vertex(graph_engine &g, page_vertex &v);
  // process a message (runs in memory)
  void run_on_message(graph_engine &g, vertex_message &msg);
  // run at the end of an iteration when all active vertices
  // in the iteration are processed.
  void run_on_iteration_end(graph_engine &g);
};
```

Figure 3: The programming interface of FlashGraph.

```
class bfs_vertex: public vertex {
  bool has_visited = false;

  void run(graph_engine &g) {
    if (!has_visited) {
      vertex_id_t id = g.get_vertex_id(*this);
      // Request the edge list of the vertex from SAFS
      request_vertices(&id, 1);
      has_visited = true;
    }
  }

  void run_on_vertex(graph_engine &g, page_vertex &v) {
    vertex_id_t dest_buf[];
    v.read_edges(dest_buf);
    g.activate_vertices(dest_buf, num_dests);
  }
};
```

Figure 4: Breadth-first search in FlashGraph.

- when it receives a message, FlashGraph executes its `run_on_message` method. This method is executed even if a vertex is inactive in the iteration.
- when the iteration comes to an end, FlashGraph executes its `run_on_iteration_end` method. A vertex needs to request this notification explicitly.

Given the programming interface, breadth-first search can be simply expressed as the code in Figure 4. If a vertex has not been visited, it issues a request to read its edge list in the `run` method and activates its neighbors in the `run_on_vertex` method. In this example, vertices do not receive other events.

This interface is designed for better flexibility and gives users fine-grained programmatic control. For example, a vertex has to explicitly request its own edge list so that a graph application can significantly reduce the amount of data brought to memory. Furthermore, the interface does not constrain the vertices that a vertex can communicate with or the edge lists that a vertex can request from SSDs. This flexibility allows FlashGraph to handle algorithms such as Louvain clustering [5], in which changes to the topology of the graph occur during computation. It is difficult to express such algorithms with graph frameworks in which vertices can only interact with direct neighbors.

3.4.1 Message passing

Message passing avoids concurrent data access to the state of other vertices. A semi-external memory graph engine cannot push data to other vertices by embedding data on edges like PowerGraph [11]. Writing data to other vertices directly can cause race conditions and requires atomic operations or locking for synchronization on vertex state. Message passing is a light-weight alternative for pushing data to other vertices. Although message passing requires synchronization to coordinate messages, it hides explicit synchronization from users and provides a more user-friendly programming interface. Furthermore, we can bundle multiple messages in a single packet to reduce synchronization overhead.

We implement a customized message passing scheme for vertex communication in FlashGraph. The worker threads send and receive messages on behalf of vertices and buffer messages to improve performance. To reduce memory consumption, we process messages and pass them to vertices when the buffer accumulates a certain number of messages.

FlashGraph supports multicast to avoid unnecessary message duplication. It is common that a vertex needs to send the same message to many other vertices. In this case point-to-point communication causes unnecessary message duplication. With multicast, FlashGraph simply copies the same message once to each thread. We implement vertex activation with multicast since activation messages contain no data and are identical.

3.5 Data representation in FlashGraph

FlashGraph uses compact data representations both in memory and on SSDs. A smaller in-memory data representation allows us to process a larger graph and use a larger SAFS page cache to improve performance. A smaller data representation on SSDs allows us to pull more edge lists from SSDs in the same amount of time, resulting in better performance.

3.5.1 In-memory data representation

FlashGraph maintains the following data structures in memory: *(i)* a graph index for accessing edge lists on SSDs; *(ii)* user-defined algorithmic vertex state of all vertices; *(iii)* vertex status used by FlashGraph; *(iv)* per-thread message queues. To save space, we choose to compute some vertex information at runtime, such as the location of an edge list on SSDs and vertex ID.

The graph index stores a small amount of information for each edge list and compute their location and size at runtime (Figure 5). Storing both the location and size in memory would require a significant amount of memory: 12 bytes per vertex in an undirected graph and 24 bytes in a directed graph. Instead, for almost all vertices, we can use one byte to store the vertex degree for an undirected vertex and two bytes for a directed vertex. Knowing the vertex degree, we can compute the edge list size and further compute their locations, since edge lists on SSDs are sorted by vertex ID. To balance computation overhead and memory space, we store the locations of a small number of edge lists in memory. By default, we store one location for every 32 edge lists, which makes computation overhead almost unnoticeable while the amortized memory overhead is small. In addition, we store the degree of large vertices (≥ 255) in a hash table. Most real-world graphs follow the power-law distribution in vertex degree, so there are only a small number of vertices in the hash table. In our default configuration, each vertex in the index uses slightly more than 1.25 bytes in an undirected graph and slightly more than 2.5 bytes in a directed graph.

Users define algorithmic vertex state in vertex programs. The semi-external memory execution model requires the size of vertex state to be a small constant so FlashGraph can keep it in memory throughout execution. In our experience, the algorithmic vertex state is usually small. For example, breadth-first search only needs one byte for each vertex (Figure 4). Many graph algorithms we implement use no more than eight bytes for each vertex. Many graph algorithms need to access the vertex ID that vertex state belongs to in a vertex program. Instead of storing the vertex ID with vertex state, we compute the vertex ID based on the address of the vertex state in memory. It is cheap to compute vertex ID most of the time. It becomes relatively more expensive to compute when FlashGraph starts to balance load because FlashGraph needs to search multiple partitions for the vertex state (Section 3.8.1).

3.5.2 External-memory data representation

FlashGraph stores edges and edge attributes of vertices on SSDs. To amortize the overhead of constructing a graph for analysis in FlashGraph and reduce SSD wearout, we use a single external-memory data structure for all graph algorithms supported by FlashGraph. Since SSDs are still several times slower than RAM, the external-memory data representation in FlashGraph has to be compact to reduce the amount of data accessed from SSDs.

Figure 5 shows the data representation of a graph on SSDs. An edge list has a header, edges and edge attributes. Edge attributes are stored separately from edges so that graph applications avoid reading attributes when they are not required. This strategy is already successfully employed by many database systems [1]. All of the edge lists stored on SSDs are ordered by vertex ID, given by the input graph.

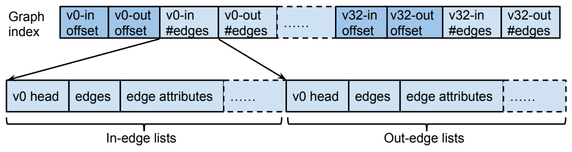


Figure 5: The data representation of a directed graph in FlashGraph. During computation, the graph index is maintained in memory and the in-edge and out-edge lists are accessed from SSDs.

FlashGraph stores the in-edge and out-edge list of a vertex separately for a directed graph. Many graph applications require only one type of edge. As such, storing both in-edges and out-edges of a vertex together would cause FlashGraph to read more data from SSDs. If a graph algorithm does require both in-edges and out-edges of vertices, having separate in-edge and out-edge lists could potentially double the number of I/O requests. However, FlashGraph merges I/O requests (Section 3.6), which significantly alleviates this problem.

3.6 Edge list access on SSDs

Graph algorithms exhibit varying I/O access patterns in the semi-external memory computation model. The most prominent is that each vertex accesses only its own edge list. In this category, graph algorithms such as PageRank [7] access all edge lists of a graph in an iteration; graph traversal algorithms require access to many edge lists in some of their iterations on most real-world graphs. A less common category of graph algorithms, such as triangle counting, require a vertex to access the edge lists of many other vertices as well. FlashGraph supports all of these access patterns and optimizes them differently.

Given the good random I/O performance of SSDs, FlashGraph selectively accesses the edge lists required by graph algorithms. Most graph algorithms only need to access a subset of edge lists within an iteration. External-memory graph engines such as GraphChi [16] and X-Stream [23] that sequentially access *all* edge lists in each iteration waste I/O bandwidth despite avoiding random I/O access. Selective access is superior to sequentially accessing the entire graph in each iteration and significantly reduces the amount of data read from SSDs.

FlashGraph merges I/O requests to maximize its performance. During an iteration of most algorithms, there are a large number of vertices that will likely request many edge lists from SSDs. Given this, it is likely that multiple edge lists required are stored nearby on SSDs, giving us the opportunity to merge I/O requests.

FlashGraph globally sorts and merges I/O requests issued by all *active state* vertices for applications where

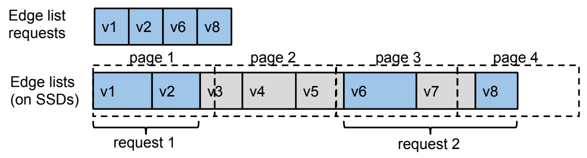


Figure 6: FlashGraph accesses edge lists and merges I/O requests.

each vertex requests a single edge list within an iteration. FlashGraph relies on its vertex scheduler (Section 3.7) to order all I/O requests within the iteration. We only merge I/O requests when they access either the same page or adjacent pages on SSDs. To minimize the amount of data brought from SSDs, the minimum I/O block size issued by FlashGraph is one flash page (4KB). As a result, an I/O request issued by FlashGraph varies from as small as one page to as large as many megabytes to benefit graph algorithms with various I/O access patterns.

Figure 6 illustrates the process of selectively accessing edge lists on SSDs and merging I/O requests. In this example, the graph algorithm requests the in-edge lists of four vertices: `v1, v2, v6 and v8`. FlashGraph issues I/O requests to access these edge lists from SSDs. Due to our merging criteria, FlashGraph merges I/O requests for `v1` and `v2` into a single I/O request because they are on the same page, and merges `v6` and `v8` into a single request because they are on adjacent pages. As a result, FlashGraph only needs to issue two, as opposed to four, I/O requests to access four edge lists in this example.

In the less common case that a vertex requests edge lists of multiple vertices, FlashGraph must observe I/O requests issued by all *running state* vertices before sorting them. In this case, FlashGraph can no longer rely on its vertex scheduler to reorder I/O requests in an iteration. The more requests FlashGraph observes, the more likely it is to merge them and generate cache hits. FlashGraph is only able to observe a relatively small number of I/O requests, compared to the size of a graph, due to the memory constraint. It is in this less common case that FlashGraph relies on SAFS to merge I/O requests to reduce memory consumption. Finally, to further increase I/O merging and cache hit rates, FlashGraph uses a flexible vertical graph partitioning scheme (Section 3.8).

3.7 Vertex scheduling

Vertex scheduling greatly affects the performance of graph algorithms. Intelligent scheduling accelerates the convergence rate and improves I/O performance. FlashGraph's default scheduler aims to decrease the number of I/O accesses and increase page cache hit rates. FlashGraph also allows users to customize the vertex scheduler to optimize for the I/O access pattern and accelerate the convergence of their algorithms. For example, scan statistics [26] in Section 4 requires large-degree vertices to be scheduled first to skip expensive computation on the majority of vertices.

FlashGraph deploys a per-thread vertex scheduler. Each thread schedules vertices in its own partition independently. This strategy simplifies implementation and results in framework scalability. The per-thread scheduler keeps multiple active vertices in the running state so that FlashGraph can observe then merge many I/O requests issued by vertex programs. In general, FlashGraph favors a large number of *running state* vertices because it allows FlashGraph to merge more I/O requests to improve performance. In practice, performance improvement is no longer noticeable past 4000 *running state* vertices per thread.

The default scheduler processes vertices ordered by vertex ID. This scheduling maximizes merging I/O requests for most graph algorithms because vertices request their own edge lists in most graph algorithms and edge lists are ordered by vertex ID on SSDs. For algorithms in which vertex ordering does not affect the convergence rate, the default scheduler alternates the direction that it scans the queue of active vertices between iterations. This strategy results in pages accessed at the end of the previous iteration being accessed at the beginning of the current iteration, potentially increasing the cache hit rate.

3.8 Graph partitioning

FlashGraph partitions a graph in two dimensions at runtime (Figure 7), inspired by two-dimension matrix partitioning. It assigns each vertex to a partition for parallel processing, shown as *horizontal partitioning* in Figure 7. FlashGraph applies the horizontal partitioning in all graph applications. In addition, it provides a flexible runtime edge list partitioning scheme within a horizontal partition, shown as *vertical partitioning* in Figure 7. This scheme, when coupled with the vertex scheduling, can increase the page cache hit rate for applications that require a vertex to access the edge lists of many vertices because this increases the possibility that multiple threads share edge list data in the cache by accessing the same edge lists concurrently.

FlashGraph assigns a worker thread to each horizontal partition to process vertices in the partition independently. The worker threads are associated with specific hardware processors. When a thread processes vertices in its own partition, all memory accesses to the vertex state are localized to the processor. As such, our partitioning scheme maximizes data locality in memory access within each processor.

FlashGraph applies a *range partitioning* function to

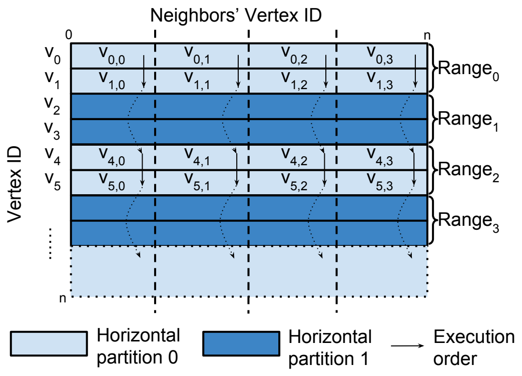


Figure 7: An example of 2D partitioning on a graph of n vertices, visualized as an adjacency matrix. In this example, the graph is split into two horizontal partitions and four vertical partitions. The size of a range in a horizontal partition is two. $v_{i,j}$ represents vertical partition j of vertex i. The arrows show the order in which the vertical partitions of vertices in horizontal partition 0 are executed in a worker thread.

horizontally partition a graph. The function performs a right bit shift on a vertex ID by a predefined number r and takes the modulo of the shifted result:

```
range_id = vid >> r
partition_id = range_id % n
```

As such, a partition consists of multiple vertex ID ranges and the size of a range is determined by a tunable parameter r. n denotes the number of partitions. All vertices in a partition are assigned to the same worker thread.

Range partitioning helps FlashGraph to improve spatial data locality for disk I/O in many graph applications. FlashGraph uses a per-thread vertex scheduler (Section 3.7) that optimizes I/O based on its local knowledge. With range partitioning, the edge lists of most vertices in the same partition are located adjacently on SSDs, which helps the per-thread vertex scheduler issue a single large I/O request to access many edge lists. The range size needs to be at least as large as the number of vertices being processed in parallel in a thread. However, a very large range may cause load imbalance because it is difficult to distribute a small number of ranges to worker threads evenly. We observe that FlashGraph works well for a graph with over 100 million vertices when r is between 12 and 18.

The vertical partitioning in FlashGraph allows programmers to split large vertices into small parts at runtime. FlashGraph replicates vertex state of vertices that require vertical partitioning and each copy of the vertex state is referred to as a *vertex part*. A user has complete freedom to perform computation on and request edge lists for a *vertex part*. In an iteration, the default FlashGraph scheduler executes all active *vertex parts* in the first vertical partition and then proceeds to the second one and so on. To avoid concurrent data access to vertex state, a *vertex part* communicates with other vertices through message passing.

The vertical partitioning increases page cache hits for applications that require vertices to access the edge lists of their neighbors. In these applications, a user can partition the edge list of a large vertex and assign a *vertex part* with part of the edge list. For example, in Figure 7, vertex v_0 is split into four parts: $v_{0,0}$, $v_{0,1}$, $v_{0,2}$ and $v_{0,3}$. Each part $v_{0,j}$ is only responsible for accessing the edge lists of its neighbors with vertex ID between $\frac{n}{4} \times j$ and $\frac{n}{4} \times (j+1)$. When the scheduler executes *vertex parts* in vertical partition j, only edge lists of vertices with vertex ID between $\frac{n}{4} \times j$ and $\frac{n}{4} \times (j+1)$ are accessed from SSDs, thus increasing the likelihood that an edge list being accessed is in the page cache.

3.8.1 Load balancing

FlashGraph provides a dynamic load balancer to address the computational skew created by high degree vertices in scale-free graphs. In an iteration, each worker thread processes active vertices in its own partition. Once a thread finishes processing all active vertices in its own partition, it 'steals' active vertices from other threads and processes them. This process continues until no threads have active vertices left in the current iteration.

Vertical partitioning assists in load balancing. FlashGraph does not execute computation on a vertex simultaneously in multiple threads to avoid concurrent data access to the state of a vertex. In the applications where only a few large vertices dominate the computation of the applications, vertical partitioning breaks these large vertices into parts so that FlashGraph's load balancer can move computation of vertex parts to multiple threads, consequently leading to better load balancing.

4 Applications

We evaluate FlashGraph's performance and expressiveness with both basic and complex graph algorithms. These algorithms exhibit different I/O access patterns from the perspective of the framework, providing a comprehensive evaluation of FlashGraph.

Breadth-first search (BFS): It starts with a single active vertex that activates its neighbors. In each subsequent iteration, the active and unvisited vertices activate their neighbors for the next iteration. The algorithm proceeds until there are no active vertices left. This requires only out-edge lists.

Betweenness centrality (BC): We compute BC by performing BFS from a vertex, followed by a back propagation [6]. For performance evaluation, we perform this process from a single source vertex. This requires both in-edge and out-edge lists.

PageRank (PR) [7]: In our PR, a vertex sends the delta of its most recent PR update to its neighbors who then update thier own PR accordingly [30]. In PageRank, vertices converge at different rates. As the algorithm proceeds, fewer and fewer vertices are activated in an iteration. We set the maximal number of iterations to 30, matching the value used by Pregel [20]. This requires only out-edge lists.

Weakly connected components (WCC): WCC in a directed graph is implemented with label propagation [33]. All vertices start in their own components, broadcast their component IDs to all neighbors, and adopt the smallest IDs they observe. A vertex that does not receive a smaller ID does nothing in the next iteration. This requires both in-edge and out-edge lists.

Triangle counting (TC) [28]: A vertex computes the intersection of its own edge list and the edge list of each neighbor to look for triangles. We count triangles on only one vertex in a potential triangle and this vertex then notifies the other two vertices of the existence of the triangle via message passing. This requires both in-edge and out-edge lists.

Scan statistics (SS) [26]: The SS metric only requires finding the maximal locality statistic in the graph, which is the maximal number of edges in the neighborhood of a vertex. We use a custom FlashGraph user-defined vertex scheduler that begins computation on vertices with the largest degree first. With this scheduler, we avoid actual computation for many vertices resulting in a highly optimized implementation [27]. This requires both in-edge and out-edge lists.

These algorithms fall into three categories from the perspective of I/O access patterns. (1) BFS and betweenness centrality only perform computation on a subset of vertices in a graph within an iteration, thus they generate many random I/O accesses. (2) PageRank and (weakly) connected components need to process all vertices at the beginning, so their I/O access is generally more sequential. (3) Triangle counting and scan statistics require a vertex to read many edge lists. These two graph algorithms are more I/O intensive than the others and generate many random I/O accesses.

5 Experimental Evaluation

We evaluate FlashGraph's performance on the applications in section 4 on large real-world graphs. We compare the performance of FlashGraph with its in-memory implementation as well as other in-memory graph engines (PowerGraph [11] and Galois [21]). For in-memory FlashGraph, we replace SAFS with in-memory data structures for storing edge lists. We also compare semi-external memory FlashGraph with external-memory graph engines (GraphChi [16] and X-Stream [23]). We further demonstrate the scalability of FlashGraph on a web graph of 3.4 billion vertices and 129 billion edges. We also perform experiments to justify some of our design decisions that are critical to achieve performance. Throughout all experiments, we use 32 threads for all graph processing engines.

Graph datasets	# Vertices	# Edges	Size	Diameter
Twitter [15]	42M	1.5B	13GB	23
Subdomain [29]	89M	2B	18GB	30
Page [29]	3.4B	129B	1.1TB	650

Table 1: Graph data sets. These are directed graphs and the diameter estimation ignores the edge direction.

We conduct all experiments on a non-uniform memory architecture machine with four Intel Xeon E5-4620 processors, clocked at 2.2 GHz, and 512 GB memory of DDR3-1333. Each processor has eight cores. The machine has three LSI SAS 9207-8e host bus adapters (HBA) connected to a SuperMicro storage chassis, in which 15 OCZ Vertex 4 SSDs are installed. The 15 SSDs together deliver approximately $900,000$ reads per second, or around $60,000$ reads per second per SSD. The machine runs Linux kernel v3.2.30.

We use the real-world graphs in Table 1 for evaluation. The largest graph is the page graph with 3.4 billion vertices and 129 billion edges. Even the smallest graph we use has 42 million vertices and 1.5 billion edges. The page graph is clustered by domain, generating good cache hit rates for some graph algorithms.

5.1 FlashGraph: in-memory vs. semi-external memory

We compare the performance of FlashGraph in semi-external memory with that of its in-memory implementation to measure the performance loss caused by accessing edge lists from SSDs.

FlashGraph scales by using semi-external memory on SSDs while preserving up to 80% performance of its in-memory implementation (Figure 8). In this experiment, FlashGraph uses a page cache of 1GB and has low cache hit rates in most applications. BC, WCC and PR perform the best and have only small performance degradation when running in external memory. Even in the worst cases, external-memory BFS and TC realize more than 40% performance of their in-memory counterparts on the subdomain Web graph.

Given around a million IOPS from the SSD array, we observe that most applications saturate CPU before saturating I/O. Figure 9 shows the CPU and I/O utilization of

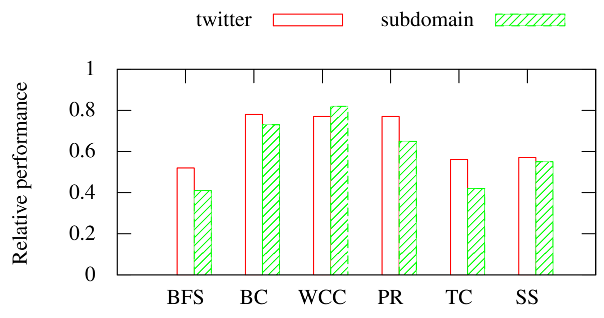


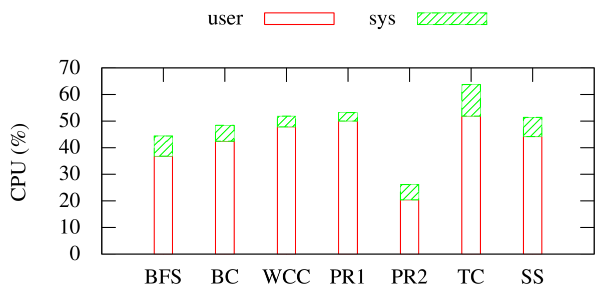
Figure 8: The performance of each application run on semi-external memory FlashGraph with 1GB cache relative to in-memory FlashGraph.

our applications in semi-external memory on the subdomain Web graph. Our machine has hyper-threading enabled, which results in 64 hardware threads in a 32-core machine, so 32 CPU cores are actually saturated when the CPU utilization gets to 50%. Both PageRank and WCC have very sequential I/O and are completely bottlenecked by the CPU at the beginning. Triangle counting saturates both CPU and I/O. It generates many small I/O requests and consumes considerable CPU time in the kernel space (almost 8 CPU cores). BFS generates very high I/O throughput in terms of bytes per second but has low CPU utilization, which suggests BFS is most likely bottlenecked by I/O. Although betweenness centrality has exactly the same I/O access pattern as BFS, it has lower I/O throughput and higher CPU utilization because it requires more computation than BFS. As a result, betweenness centrality is bottlenecked by CPU most of the time. The CPU-bound applications tend to have a small performance gap between in-memory and semi-external memory implementations.

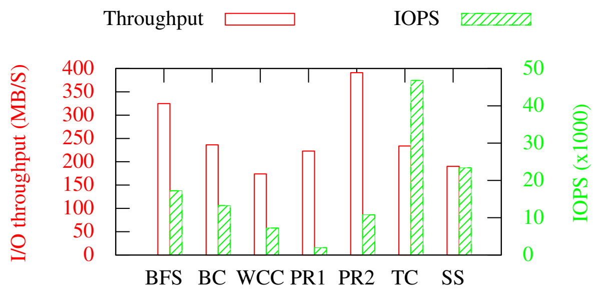
5.2 FlashGraph vs. in-memory engines

We compare the performance of FlashGraph to PowerGraph [11], a popular distributed in-memory graph engine, and Galois [21], a state-of-art in-memory graph engine. FlashGraph and Powergraph provide a general high-level vertex-centric programming interface, whereas Galois provides a low-level programming abstraction for building graph engines. We run these three graph engines on the Twitter and subdomain Web graphs. Unfortunately, the Web page graph is too large for in-memory graph engines. We run PowerGraph in multi-thread mode to achieve its best performance and use its synchronous execution engine because it performs better than the asynchronous one on both graphs.

Both in-memory and semi-external memory FlashGraph performs comparably to Galois, while signif-

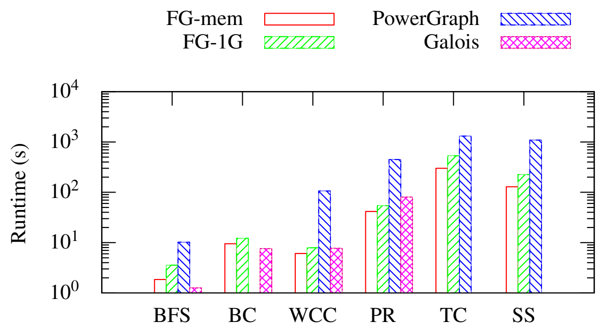


(a) CPU utilization in the user and kernel space. Hyper-threading enables 64 hardware threads in a 32-core machine, so 50% CPU utilization means 32 CPU cores are saturated.

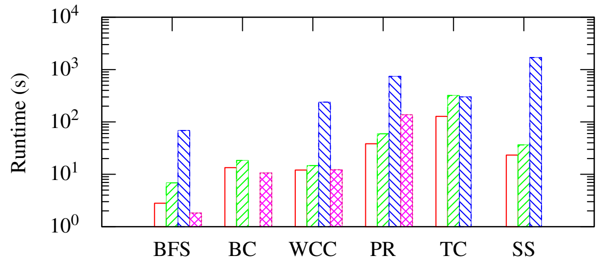


(b) I/O utilization.

Figure 9: CPU and I/O utilization of FlashGraph on the subdomain Web graph. PR1 is the first 15 iterations of PageRank and PR2 is the last 15 iterations of PageRank.



(a) On the Twitter graph.



(b) On the subdomain Web graph.

Figure 10: The runtime of different graph engines. FG-mem is in-memory FlashGraph. FG-1G is semi-external memory FlashGraph with a page cache of 1 GB.

icantly outperforming PowerGraph (Figure 10). In-memory FlashGraph outperforms Galois in WCC and PageRank. It performs worse than Galois in graph traversal applications such as BFS and betweenness centrality, because Galois uses a different algorithm [3] for BFS. The algorithm reduces the number of edges traversed in both applications. The same algorithm could be implemented in FlashGraph but would not benefit semi-external memory FlashGraph because the algorithm requires access to both in-edge and out-edge lists, thus, significantly increasing the amount of data read from SSDs.

5.3 FlashGraph vs. external memory engines

We compare the performance of FlashGraph to that of two external-memory graph engines, X-Stream [23] and GraphChi [16]. We run FlashGraph in semi-external memory and use a 1 GB page cache. We construct a software RAID on the same SSD array to run X-Stream and GraphChi. Note that GraphChi does not provide a BFS implementation, and X-Stream implements triangle counting via a semi-streaming algorithm [4].

FlashGraph outperforms GraphChi and X-Stream by one or two orders of magnitude (Figure 11a). FlashGraph only needs to access the edge lists and performs computation on only the vertices required by the graph application. Even though FlashGraph generates random I/O accesses, it saves both CPU and I/O by avoiding unnecessary computation and data access. In contrast, GraphChi and X-Stream sequentially read the entire graph dataset multiple times.

Although FlashGraph uses its semi-external memory mode, it consumes a reasonable amount of memory when compared with GraphChi and X-Stream (Figure 11b). In some applications, FlashGraph even has smaller memory footprint than GraphChi. FlashGraph's small memory footprint allows it to run on regular desktop computers, comfortably processing billion-edge graphs.

5.4 Scale to billion-node graphs

We further evaluate the performance of FlashGraph on the billion-scale page graph in Table 1. FlashGraph uses a page cache of 4GB for all applications. To the best of our knowledge, the page graph is the largest graph used for evaluating a graph processing engine to date. The closest one is the random graph used by Pregel [20], which has a billion vertices and 127 billion edges. Pregel processed it on 300 multicore machines. In contrast, we process the page graph on a single multicore machine.

FlashGraph can perform all of our applications within a reasonable amount of time and with relatively small memory footprint (Table 2). For example, FlashGraph achieves good performance in BFS on this billion-node graph. It takes less than five minutes with a cache size of 4GB; i.e., FlashGraph traverses nearly seven million vertices per second on the page graph, which is much higher than the maximal random I/O performance (900,000 IOPS) provided by the SSD array. In contrast, Pregel [20] used 300 multicore machines to run the shortest path algorithm on their largest random graph and took a little over ten minutes. More recently, Trinity [24] took over ten minutes to perform BFS on a graph of one billion vertices and 13 billion edges on 14 12-core machines.

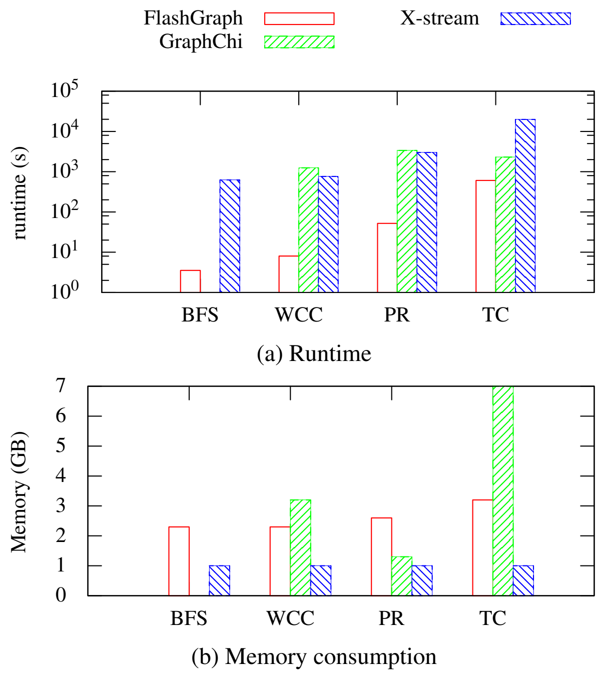


Figure 11: The runtime and memory consumption of semi-external memory FlashGraph and external memory graph engines on the Twitter graph.

Our solution allows us to process a graph one order of magnitude larger than the page graph on a single commodity machine with half a terabyte of RAM. The maximal graph size that can be processed by FlashGraph is limited by the capacity of RAM and SSDs. Our current hardware configuration allows us to attach 24 1TB SSDs to a machine, which can store a graph with over one trillion edges. Furthermore, the small memory footprint suggests that FlashGraph is able to process a graph with tens of billions of vertices.

FlashGraph results in a more economical solution to process a massive graph. In contrast, it is much more expensive to build a cluster or a supercomputer to process a graph of the same scale. For example, it requires 48 machines with 512GB RAM each to achieve 24TB aggregate RAM capacity, so the cost of building such a cluster is at least 24 – 48 times higher than our solution.

Algorithm	Runtime (sec)	Init time (sec)	Memory (GB)
BFS	298	30	22
BC	595	33	81
TC	7818	31	55
WCC	461	32	47
PR	2041	33	46
SS	375	58	83

Table 2: The runtime and memory consumption of FlashGraph on the page graph using a 4GB cache size.

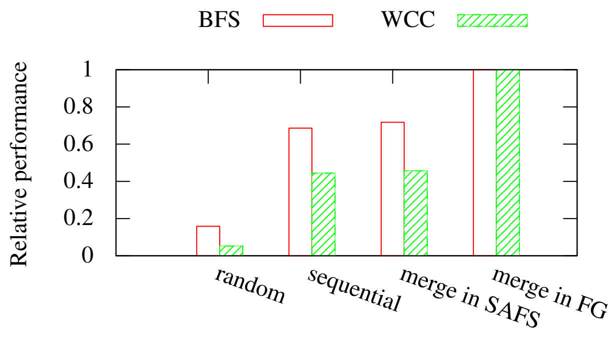


Figure 12: The impact of preserving sequential I/O access in graph applications. All performance is relative to that of the implementation of merging I/O requests in FlashGraph.

In addition, FlashGraph minimizes SSD wearout and the only write required by FlashGraph is to load a new graph to SSDs for processing. Therefore, we can further reduce the hardware cost, by using consumer SSDs instead of enterprise SSDs to store graphs, as well as reducing the maintenance cost.

5.5 The impact of optimizations

In this section, we perform experiments to justify some of our design decisions that are critical to achieve performance for FlashGraph in semi-external memory.

5.5.1 Preserve sequential I/O

We demonstrate the importance of taking advantage of sequential I/O access in graph applications, using BFS and weakly connected components. We start with vertex execution performed in random order, and then sequentially order vertex execution by vertex ID. Finally, we show the performance difference between merging I/O requests in SAFS vs. FlashGraph. All experiments are run on the subdomain web graph.

The huge gap (Figure 12) between random execution and sequential execution suggests that there exists a degree of sequential I/O in both applications, as described in Section 3.6. If FlashGraph did not take advantage of these sequential I/O accesses, it would suffer substantial performance degradation. Therefore, the first priority of the vertex scheduler in FlashGraph is to schedule vertex execution to generate sequential I/O. Consequently, FlashGraph's vertex scheduler is highly constrained by I/O ordering requirements and is not able to schedule vertex execution freely like Galois [21].

Figure 12 also shows that I/O accesses generated by a graph algorithm are well merged in FlashGraph as opposed to the filesystem level or the block subsystem level. Although SAFS, the Linux filesystem and the Linux block subsystem are capable of merging I/O requests, they require more CPU computation to merge I/O requests and do not have a global view for merging I/O requests. Consequently, it is much more light-weight and effective to merge I/O requests in FlashGraph. By doing so, we achieve 40% speedup for BFS and more than 100% speedup for WCC.

5.5.2 The impact of the page size

In this section, we investigate the impact of the page size in SAFS. A page is the smallest I/O block that FlashGraph can access from SSDs. The experiments are run on the subdomain web graph.

Figure 13 shows that FlashGraph should use 4KB as the SAFS page size. SSDs store and access data at the granularity of 4KB flash pages, so using an SAFS page smaller than 4KB does not increase the I/O rate of SSDs much. A larger SAFS page size brings in more unnecessary data and wastes I/O bandwidth, which leads to performance degradation. When we increase the SAFS page size from 4KB to 1MB, the performance of BFS and triangle counting (TC) decreases to a small fraction of their maximal performance. Even WCC, whose I/O access is more sequential, performs better with 4KB pages because WCC also needs to selectively access edge lists in all iterations but the first. This result suggests that TurboGraph [12], which uses a block size of multiple megabytes, may perform general graph analysis suboptimally. It also suggests that when using 4KB pages, selectively accessing edge lists and merging I/O enables FlashGraph to adapt to different I/O access patterns.

5.6 The impact of page cache size

We investigate the effect of the SAFS page cache size on the performance of FlashGraph. We vary the cache size from 1 GB to 32GB, which is sufficiently large to accommodate the twitter graph and the subdomain web graph. We omit Twitter graph results as they mirror subdomain graph results.

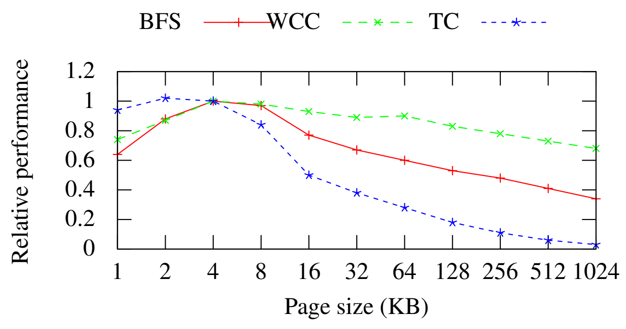


Figure 13: The impact of the page size in FlashGraph. All performance is relative to that of the implementation with 4KB page size.

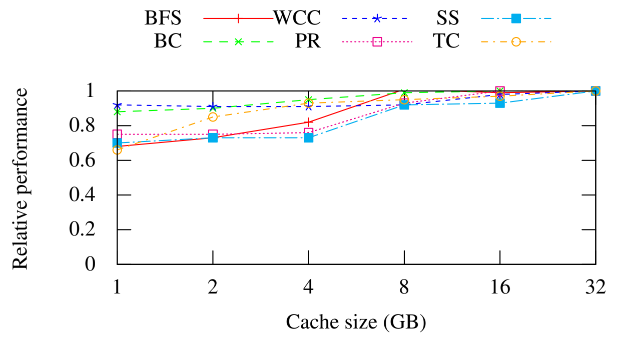


Figure 14: The impact of cache size in FlashGraph.

FlashGraph performs well even with a small page cache (Figure 14). With a 1GB page cache, all applications realize at least 65% of their performance with a 32GB page cache, and WCC and betweenness centrality even achieve around 90% of the performance with a 32GB page cache. Although PageRank has a similar I/O access pattern to WCC, it converges more slowly than WCC, so a large cache has more impact on PageRank. By varying the page cache size, we show FlashGraph can smoothly transition from a semi-external memory graph engine to an in-memory graph engine.

6 Conclusions

We present the semi-external memory graph engine called FlashGraph that closely integrates with an SSD filesystem to achieve maximal performance. It uses an asynchronous user-task I/O interface to reduce overhead associated with accessing data in the filesystem and overlap computation with I/O. FlashGraph selectively accesses edge lists required by a graph algorithm from SSDs to reduce data access; it conservatively merges I/O requests to increase I/O throughput and reduce CPU consumption; it further schedules the order of processing vertices to help merge I/O requests and maximize the page cache hit rate. All of these designs maximize performance for applications with different I/O access patterns. We demonstrate that a semi-external memory graph engine can achieve performance comparable to in-memory graph engines.

We observe that in many graph applications a large SSD array is capable of delivering enough I/Os to saturate the CPU. This suggests the importance of optimizing for CPU and RAM in such an I/O system. It also suggests that SSDs have been sufficiently fast to be an important extension for RAM when we build a machine for large-scale graph analysis applications.

FlashGraph provides a concise and flexible programming interface to express a wide variety of graph algorithms and their optimizations. Users express graph algorithms in FlashGraph from the perspective of vertices. Vertices can interact with any other vertices in the graph by sending messages, which localizes user computation to the local memory and avoids concurrent data access to algorithmic vertex state.

Unlike other external-memory graph engines such as GraphChi and X-stream, FlashGraph supports selective access to edge lists. We demonstrate that streaming the entire graph to reduce random I/O leads to a suboptimal solution for high-speed SSDs. Reading and computing on data only required by graph applications saves computation and increases the I/O access rate to the SSDs.

We further demonstrate that FlashGraph is able to process graphs with billions of vertices and hundreds of billions of edges on a single commodity machine. FlashGraph, on a single machine, meets and surpasses the performance of distributed graph processing engines that run on large clusters. Furthermore, the small memory footprint of FlashGraph suggests that it can handle a much larger graph in a single commodity machine. Therefore, FlashGraph results in a much more economical solution for processing massive graphs, which makes massive graph analysis more accessible to users and provides a practical alternative to large clusters for such graph analysis.

7 Acknowledgments

We would like to thank the FAST reviewers, especially our shepherd John Ousterhout, for their insightful comments and guidance for revising the paper. We also thank Heng Wang and Vince Lyzinski in Department of Applied Mathematics & Statistics of the Johns Hopkins University for discussing the applicatins of FlashGraph. This work is supported by DARPA N66001-14-1-4028, NSF ACI-1261715, NIH NIBIB 1RO1EB016411-01, and DARPA XDATA FA8750-12-2-0303.

References

[1] Abadi, D., Boncz, P., Harizopoulos, S., Idreos, S., and Madden, S. The design and implementation of modern column-oriented database systems. Foundations and Trends in Databases 5 (2013), 197–280.

[2] Abello, J., Buchsbaum, A. L., and Westbrook, J. R. A functional approach to external graph algorithms. In Algorithmica (1998), Springer-Verlag, pp. 332–343.

[3] Beamer, S., Asanović, K., and Patterson, D. Direction-optimizing breadth-first search. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (2012), SC '12.

[4] Becchetti, L., Boldi, P., Castillo, C., and Gionis, A. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (2008).

[5] Blondel, V. D., loup Guillaume, J., Lambiotte, R., and Lefebvre, E. Fast unfolding of communities in large networks. Journal of Statistical Mechanics: Theory and Experiment (2008).

[6] Brandes, U. A faster algorithm for betweenness centrality. Journal of Mathematical Sociology 25 (2001), 163–177.

[7] Brin, S., and Page, L. The anatomy of a large-scale hypertextual web search engine. In Proceedings of the Seventh International Conference on World Wide Web 7 (1998).

[8] Chen, R., Weng, X., He, B., Yang, M., Choi, B., and Li, X. On the efficiency and programmability of large graph processing in the cloud. Tech. rep., Microsoft Research, 2010.

[9] Dean, J., and Ghemawat, S. MapReduce: Simplified data processing on large clusters. In Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6 (2004).

[10] Apache giraph. https://giraph.apache.org/, Accessed 4/9/2014.

[11] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. PowerGraph: Distributed graph-parallel computation on natural graphs. In Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (2012).

[12] 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.

[13] Kang, U., Tsourakakis, C. E., and Faloutsos, C. PEGASUS: A peta-scale graph mining system implementation and observations. In Proceedings of the 2009 Ninth IEEE International Conference on Data Mining (2009).

[14] Kepner, J., and Gilbert, J. Graph Algorithms in the Language of Linear Algebra. Society for Industrial & Applied Mathematics, 2011.

[15] Kwak, H., Lee, C., Park, H., and Moon, S. What is twitter, a social network or a news media? In Proceedings of the 19th International Conference on World Wide Web (2010).

[16] Kyrola, A., Blelloch, G., and Guestrin, C. Graphchi: Large-scale graph computation on just a PC. In Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (2012).

[17] Leskovec, J., Lang, K. J., Dasgupta, A., and Mahoney, M. W. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. Internet Mathematics 6, 1 (2009), 29–123.

[18] 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. (2012).

[19] Lugowski, A., Alber, D., Bulu, A., Gilbert, J., Reinhardt, S., Teng, Y., and Waranis, A. A flexible open-source toolbox for scalable complex graph analysis. In Proceedings of the 2012 SIAM International Conference on Data Mining (2012).

[20] 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).

[21] 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 (2013).

[22] Pearce, R., Gokhale, M., and Amato, N. M. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (2010).

[23] 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).

[24] 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).

[25] Shun, J., and Blelloch, G. E. Ligra: A lightweight graph processing framework for shared memory. In Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2013).

[26] Wang, H., Tang, M., Park, Y., and Priebe, C. E. Locality statistics for anomaly detection in time series of graphs. IEEE Transactions on Signal Processing 62, 3 (2014).

[27] Wang, H., Zheng, D., Burns, R., and Priebe, C. Active community detection in massive graphs. CoRR abs/1412.8576 (2015).

[28] Watts, D. J., and Strogatz, S. H. Collective dynamics of 'small-world' networks. Nature 393, 440-442 (1998).

[29] Web graph. http://webdatacommons.org/hyperlinkgraph/, Accessed 4/18/2014.

[30] Zhang, Y., Gao, Q., Gao, L., and Wang, C. Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation. IEEE Transactions on Parallel and Distributed Systems (2014).

[31] Zheng, D., Burns, R., and Szalay, A. S. A parallel page cache: Iops and caching for multicore systems. In Proceedings of the 4th USENIX Conference on Hot Topics in Storage and File Systems (2012).

[32] Zheng, D., Burns, R., and Szalay, A. S. Toward millions of file system IOPS on low-cost, commodity hardware. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (2013).

[33] Zhu, X., and Ghahramani, Z. Learning from labeled and unlabeled data with label propagation. Tech. rep., Carnegie Mellon University, 2002.

Host-side Filesystem Journaling for Durable Shared Storage

Andromachi Hatzieleftheriou and Stergios V. Anastasiadis
Department of Computer Science and Engineering, University of Ioannina, Ioannina 45110, Greece

Abstract

Hardware consolidation in the datacenter occasionally leads to scalability bottlenecks due to the heavy utilization of critical resources, such as the shared network bandwidth. Host-side caching on durable media is already applied at the block level in order to reduce the load of the storage backend. However, block-level caching is often criticized for added overhead, and restricted data sharing across different hosts. During client crashes, writeback caching can also lead to unrecoverable loss of written data that was previously acknowledged as stable. We improve the durability of shared storage in the datacenter by supporting journaling at the kernel-level client of an object-based distributed filesystem. Storage virtualization at the file interface achieves clear consistency semantics across data and metadata blocks, supports native file sharing between clients over the same or different hosts, and provides flexible configuration of the time period during which the data is durably staged at the host side. Over a prototype implementation that we developed, we experimentally demonstrate improved performance up to 58% for specific durability guarantees, and reduced network and disk bandwidth at the storage servers by up to 42% and 82%, respectively.

1 Introduction

Infrastructure virtualization in the datacenter typically consolidates client and server nodes on similar hardware. Network storage is often provided by scalable server clusters through protocols operating at the file, block or object level. The file interface is attractive for its sharing and efficiency properties [30, 13, 25, 40, 6, 22, 1]; the block interface provides convenient virtualization flexibility but incurs undesirable translation overheads [24, 20, 35, 36, 26]; and the object interface is scalable and efficient because it carries semantical information for specialized storage management [30, 39, 41].

Another design dimension in datacenter storage applies client-side caching for improved performance and durability at reduced network and server load. Existing solutions often apply block-level caching at the client-side host, and they adopt write-through or writeback policy according to the application and hardware characteristics. A write-through policy is preferred for read caching without data loss at device failure. Instead, a writeback policy improves the resource efficiency and application performance but makes the cache device part of the failure model [24, 34, 9, 19, 31, 16].

The Arion system is a new design point that we introduce in cloud storage to improve the durability of the file interface at the client side (Fig. 1 fully explained in Section 2). We integrate the client software of a distributed filesystem with persistent host-based storage over a journal device. We enhance the flushing functionality of the filesystem client with tunable control of both the amount of dirty pages that are staged at the host, and the time period taken by dirtied pages to reach the backend servers. At increased flushing frequency to the journal device, we practically minimize the recovery point objective (RPO) close to zero under the following condition: the dominant cause of a client crash is operator or software bug rather than permanent hardware loss, such as that of the local storage device [28, 4].

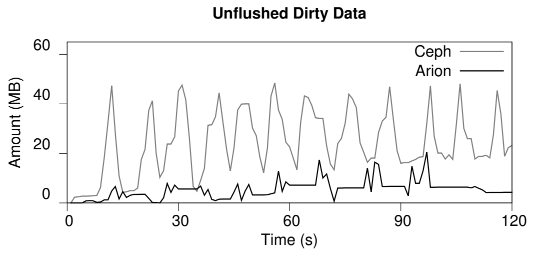


Figure 1: Dirty data that remains unflushed in the volatile memory of the client in Ceph and the proposed Arion system.

The availability and performance of cloud services depends on the ordering, durability and membership properties of replication consistency [7]. In a multi-tier system, data is replicated at the frontend application, an intermediate caching layer, and the backend persistent storage. For reduced cross-layer communication, the frontend can be stateless and lose recently written data during a crash or reboot. Recognition of this risk has urged the designers of local filesystems, flash-based caches and distributed storage systems to emphasize the ordering guarantees of crash consistency at the expense of weaker durability [10, 19, 26].

I/O-intensive workloads in a distributed filesystem can take advantage of writeback caching at the client side to improve their performance and reduce the respective network and server load. Unfortunately, current scalable filesystems can natively support only in-memory caching at the client side. This deficiency has been partially addressed by having the filesystem client running in the hypervisor and enforcing the guests to mount disk images as plain files through a block interface that enables block-based caching [9]. But this approach has been criticized for the increased overheads from the semantic gap that it causes and the unnecessary multiple translations between the file and block interface [13, 20, 35].

Our main contributions are the following: (i) We improve the durability of frontend memory caching by integrating disk-based journaling into the client of a distributed filesystem. (ii) We implement a prototype of the proposed storage layer in the kernel-level client of the Ceph object-based filesystem. (iii) We experiment with several application-level benchmarks over a virtualized host and a clustered storage backend. Our approach enables frequent flushes of dirty pages to the local journal without crossing the network and hitting the disks of the backend storage. Therefore, in a host machine with reliable local storage, we approximate the consistency ordering and durability of write-through caching with the configurable efficiency of periodic writeback.

We further motivate our research in Section 2 and describe the Arion architecture in Section 3. We present our software prototype in Section 4 and explain our experimental results in Section 5. In Sections 6 we compare our work with previous research, and in Section 7 we clarify our contributions and some limitations. Finally, in Section 8 we outline our conclusions and possible extensions.

2 Motivation

Loss or corruption of committed updates to critical data is recognized as a particularly damaging class of failure [4]. This observation is highly relevant in a large-scale multi-tier environment, with mean time between failures inversely proportional to the number of machines. Several studies conclude that hardware failures contribute much less to service-level failures in comparison to causes related to software bugs and faults from operator or maintenance tasks [28, 4].

In traditional Unix, written data is acknowledged asynchronously to the application but only flushed periodically to the local disk. This approach has been adopted by several distributed filesystems in the form of asynchronous data transfer from the volatile memory of the client to the servers [27, 23]. Although durable caching at the client side can reduce the network load of the servers, it complicates the maintenance of replication consistency among different clients or between the clients and the servers [14].

In Fig. 1, we measure the amount of dirty data that remains unflushed at the client memory over time. We compare Ceph [41] under default flushing parameters with the proposed Arion system (Section 3). In the environment of Section 5, we used the fileserver mode of Filebench [12] running for 2min over 10000 files. The Linux pdflush daemon wakes up every 5s and transfers dirty data older than 30s from the client to the servers [8]. Additionally, the Arion client every 1s flushes dirty data to the local journal of the host. On average over time, the Ceph client keeps 24.3MB of dirty data solely in volatile memory, i.e., unrecoverable from a crash. Instead, the Arion host-side journaling reduces to 5.4MB the vulnerable data in the volatile memory of the client.

3 System Architecture

Next we outline our assumptions and goals before we describe the main design ideas of Arion and consistency.

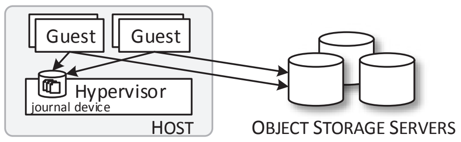


Figure 2: Host-side journaling in the Arion architecture.

3.1 Assumptions and Goals

We aim to improve the durability and performance of shared storage in the datacenter at reduced utilization of the server resources. User is the application-level entity that initiates I/O requests to the filesystem, and client is the host-based software that provides filesystem access to users. We target host hardware with reliability characteristics on par with those of the server machines. The host provides directly-attached storage with sufficient redundancy to tolerate the occasional failure of a single device. Appropriate storage technologies include hard disks, solid-state drives, or non-volatile memory. In the proposed storage architecture we aim to support the following properties:

i) **Interface** Stored data is directly accessible for regular use and maintenance tasks over the network with a POSIX-like file-based interface [37].

ii) **Sharing** Heterogeneous clients on the same or different hosts can natively share data at the storage level but may also apply synchronizations at the application level.

iii) **Durability** Most recent writes survive client reboots but require redundant hardware support to tolerate permanent failures of individual storage devices at the host.

iv) **Performance** Client writes are safely stored at sequential disk throughput, but the read performance depends on the efficiency of the client memory cache.

v) **Scalability** The storage backend linearly scales out to efficiently hold increasing amounts of data.

3.2 Design

We rely on an object-based scale-out backend of multiple data and metadata servers (Fig. 2). The client runs over either a guest system on virtualized hardware or a standalone system on bare metal. A read operation synchronously returns the latest version of the requested state. A synchronous write reaches a configurable number of durable replicas before it returns. An asynchronous write returns as soon it updates the buffer cache of the client system, but the modified blocks have to reach a configurable number of durable replicas before they are considered safely stored.

We regard the frontend logging to a persistent storage medium as a complementary form of replication. Unlike the traditional replication that is homogeneously applied across functionally equivalent backend servers, the frontend logging adds *heterogeneity* with respect to the storage format, the logical layer and the time duration of the replica[1].

[1]The Coda filesystem previously introduced the concept of two-tier replication in the context of disconnected operation [18](see also Section 6).

The metadata server (MDS) enables shared filesystem access at *file* granularity through different types of tokens leased to the clients. Supported access types include exclusive write (cached) by a single client, and concurrent read (cached) or concurrent write (uncached) by multiple clients. A client can only cache the writes of data and metadata accessed with exclusive permission. The file interface provides valuable semantical information about the consistency dependencies of modified data and metadata. When a client transfers the file updates to the servers, the metadata is written only after the referenced data blocks have safely reached the server state.

The key innovation of our design is the integration of a local journal with the kernel-level client of a distributed filesystem (Fig. 2). The host-side journal is distinct for each guest in virtualized hosts. The client inserts into the journal *both* the data and metadata modified by an I/O request. Thus we ensure that a metadata version matches the version of the data it refers to (version consistency [11]). We only keep one transaction active to accept all the (redo) records of low-level I/O operations corresponding to an atomic filesystem request. An active transaction closes as a result of timeout expiration, explicit flush request, or reclamation of journal space [8].

A journaled block remains cached in the client memory until it is safely written to the servers. If an MDS revokes the write token from a client due to some conflict (e.g., concurrent writes to the same file), the client is forced to write (checkpoint) the conflicting writes to the servers and invalidate the respective journal records. On client disconnection from the servers, the leased tokens may expire and the client will no longer be able to access the files locally [23]. At network reconnection, the client writes to the servers the mutated blocks of each file whose token has been refreshed and whose metadata cached at the client is newer than the file metadata at the MDS, but it discards the remaining blocks.

The primary benefit from host-side journaling in a distributed filesystem is the reduced vulnerability of outstanding writes in the volatile memory of the client. If a client crashes and reboots without hardware failure at the host, then the client replays the completed transactions and transfers the recorded updates to the filesystem servers. We update a file only if the replaying client confirms token ownership, and the journaled metadata is newer than the file metadata at the MDS. In case of client crash during the recovery, the replay is repeated until the client journal is fully checkpointed.

The durable storage of recent writes over the host-side journal improves the server writeback efficiency with respect to the utilized network and disk bandwidth. The consumed shared resources are reduced through batching applied to repetitive writes over the same blocks, or to small writes. At synchronous writes, we journal the updates locally and postpone the server writeback as permitted by the flushing parameter settings. Thus, performance improves depending on the pressure over the shared resources and the resulting queuing delays in the I/O path of the Arion networked storage.

3.3 Consistency

We strengthen the durability of memory-based caching in clients that provide native support for file sharing. The file interface differentiates the data blocks from the metadata. Thus, our system cleanly addresses issues of vertical (client-server) and horizontal (client-client) consistency across different replicas [9]. In the order imposed by their arrival time and structural dependencies, the data and metadata updates are first journaled at the local host and subsequently persisted at the backend servers. Additionally, the filesystem arbitrates the conflicts among different clients through lease-based tokens. In contrast, block-based schemes typically operate transparently to the filesystem, and as a result explicitly track the order and *relax* the durability of block updates [10, 19, 26].

4 System Prototype

Next we provide background information on the Linux kernel and Ceph, before we present the implementation of Arion.

4.1 Background

Linux The Linux kernel maintains in memory a page cache with data and metadata blocks of recently accessed disk files [8]. A page descriptor stores bookkeeping information about the address space and the inode of a page. For every cached disk block, there is a block buffer that stores the actual data, and a buffer head structure with bookkeeping information. The dirty pages are written to disk at timeout expiration, under space pressure in the main memory or the journal device, and by explicit flush request from the user.

The Linux kernel implements filesystem journaling with a special kernel layer, the Journaling Block Device (JBD). All the records of the low-level operations that belong to a high-level atomic update are stored in the same transaction of the journal. The journaling I/O of each block buffer is managed through a separate buffer head structure in kernel. Additionally, a journal head structure links each block buffer to the corresponding transaction. One or more journal descriptor blocks mark the beginning of the transaction and store the tags of journal blocks belonging to the transaction. A commit operation writes to the journal the dirty buffers of a transaction followed by a commit block. A checkpoint operation transfers the records of a transaction to the filesystem state and deletes the transaction from the journal.

Ceph The Ceph is an object-based parallel filesystem designed for scalability, performance and high availability [41]. It consists of four main components: the clients provide a POSIX-like I/O interface; the metadata servers (MDS) manage the namespace hierarchy; the object storage devices (OSD) reliably store data and metadata; and the monitors (MON) manage the server cluster map. A set of MDSs acts as a scalable, consistent, distributed cache of the file namespace. The metadata is persistently stored on the OSDs as a collection of regular objects. Ceph maps each object to a placement group consisting of multiple OSDs. Each

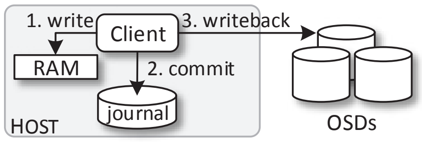


Figure 3: A write request is applied to the kernel memory, then added to the host journal and finally reaches the servers.

OSD maintains a local journal to handle object versioning and update serialization.

The kernel-level client prepares a page in the cache upon a write request. A partial page update fetches the original page to the cache from the OSDs, copies on it the user modifications and marks the inode object as dirty. The Linux pdflush threads wake up periodically to scan the list of dirty inodes and write their dirty pages to the OSDs. The writeback time refers to the wake-up period, and the expiration time refers to the time length after which a dirty page is flushed. After the writeback completion at the OSDs, the client transfers the dirty inode to the MDS and receives acknowledgment when the inode update is safely stored.

4.2 Implementation

In order to improve the durability and efficiency of shared data storage over Ceph, we increase the statefulness of the filesystem client with a local journal (Fig. 3). Our prototype implementation integrates the Linux JBD with the CephFS kernel-level client and the other Ceph components. In the `ceph_fs_client` structure of CephFS we added two extra fields: the `journal_bdev` referring to a block-device control structure in the kernel, and the `s_journal` referring to the journal control structure of JBD. We specify the journal device to the kernel with a new mount option that we added in the system.

For the addition of journaling support in Ceph, we allocate disk block buffers and buffer heads during a write with the `ceph_write_begin()` function. We also create a journal head to insert each block buffer to an active transaction. The modified pages are written to the network servers either periodically, or under pressure from the client memory and journal space. After a particular page is written back or invalidated, we also invalidate the respective journal records.

The `private` field of a page descriptor is used by the local Linux filesystem to typically link a page with a block buffer, and by Ceph to maintain the context of the supported snapshot service. Instead, we introduce an auxiliary metadata structure, called `ceph_metapage`, to associate a page with the block buffer, the snapshot context, and several inode attributes of the file. We link the above metadata structure to the page via the `private` field of the page descriptor. Arion allocates the `ceph_metapage` structure when it creates the block buffers for a page, and deallocates it after the page is written back to the servers.

In kernel, we added a new page state, called `JBD_state`, to indicate that a page has been marked for journaling but not committed yet to the journal. The journaling of metadata operations has been particularly challenging to implement in Arion, because there are several places in the I/O path of the original Ceph that mark the inode as dirty. Also, unlike local filesystems, the Ceph client does not directly cache an inode object as a raw metadata block. In order to effectively manage the file metadata in the journal of Arion, we substantially expanded the JBD tag structure in the journal descriptor block to include several inode attributes provided by the MDS.

As a result, the journal tag of Arion contains fields to identify the number of modified data blocks, the modification offset range, and the inode number, version, size, permissions and latest time of different operation types. During the crash recovery of a client, we compare the inode metadata contained in the journal tag of a file against the respective attributes freshly fetched from the MDS. Subsequently, we only replay the write requests of files whose journaled metadata has not been obsoleted in the MDS by accesses that occurred in the time period between the transaction commit and the ongoing recovery.

Our current prototype implementation fully supports (i) the journaling of mutated data and metadata from the client memory to the host-side journal, and (ii) the filesystem recovery after a client crash that leaves the host hardware operational.

5 Performance Evaluation

We implemented the Arion host-side journaling based on Linux JBD2 and the kernel-level client of Ceph (v0.80.1). The Arion development required 3417 new commented lines across 15 files of Linux kernel (v3.6.6). Next we describe our experimentation environment, and the measured performance and resource consumption of Arion and original Ceph.

5.1 Experimentation Environment

The host machine is a rack server with 2 quad-core x86-64 2.66GHz processors, 7GB RAM, 2 bonded 1GbE links, and two 300GB 15KRPM SAS HDDs in RAID0 configuration. The host uses Linux kernel v3.5.5 with Xen v4.2.0, and the guest runs Linux v3.6.6 over 2GB RAM and 2 pinned VCPUs. Arion uses a 2GB disk partition at the host for local journal. We leave for future work the study with other types of durable devices (e.g., SSDs). The guest client mounts directly the distributed filesystem, and the hypervisor provides local access to the network and journal devices.

Each of Ceph and Arion uses 5 machines: 3 OSDs, 1 MON and 1 MDS. The machine is a rack server running Linux kernel (v3.10.41) over 2 quad-core x86-64 2.66GHz processors, 3GB RAM, 1 GbE link, and two separate 300GB 15KRPM SAS HDDs. A stored object is replicated over 3 OSDs. Each OSD dedicates one disk for journaling (1GB partition).

Our experiments are based on the Filebench v1.4.9.1 macrobenchmark (fileserver, varmail, createfiles) and the FIO v2.1.7 microbenchmark. We clear the caches before each experiment. We keep the on-disk write buffers *disabled* at

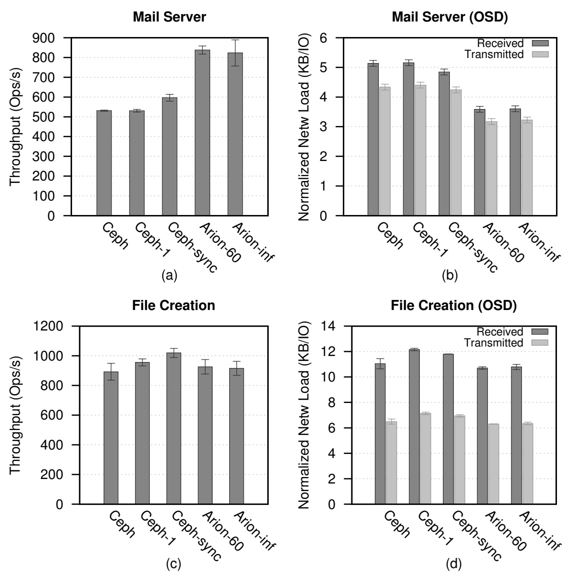


Figure 4: Operation throughput and normalized network load with the varmail (a,b) and createfiles (c,d) modes of Filebench across different settings of Ceph and Arion.

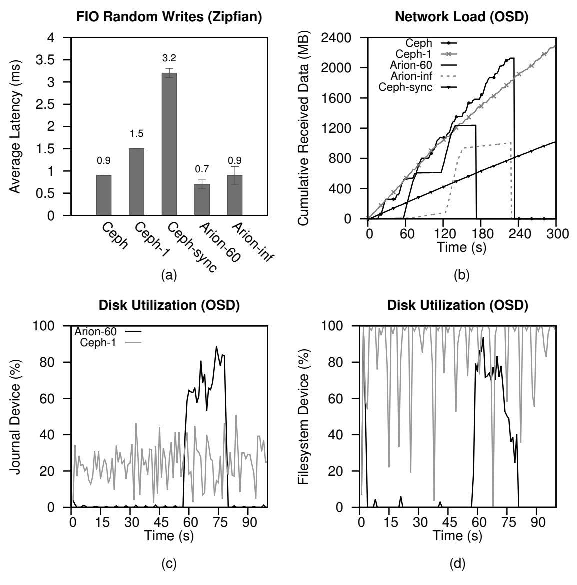


Figure 5: (a) Average latency, (b) cumulative network load at one OSD, and disk utilization at the (c) journal and (d) filesystem OSD disks across different settings of Ceph and Arion.

the host, but activated at the servers of Ceph and Arion [32]. RAID0 with two disks does not give unfair advantage to host journaling because the storage backend already consists of multiple servers with two disks each. In the shown bar charts we include 95% confidence intervals from 5 repetitions.

5.2 Measurements

In Fig. 4 we run two Filebench modes with default settings (for 5min) to study the system performance and efficiency. We examine Ceph with the writeback and expiration time respectively set to the default 5s and 30s (Ceph), or both set equal to 1s (Ceph-1), or the filesystem mounted in synchronous mode (Ceph-sync). We also examine Arion with dirty blocks periodically copied to the host-side journal every 1s, and the writeback and expiration times both set equal to 60s (Arion-60), or infinity (Arion-inf) to minimize writeback.

Varmail emulates multi-threaded I/O activity of a server *synchronously* storing email messages across 50000 files. In Fig. 4a, Arion-60 achieves operation throughput of 837.8 ops/s, or 58% higher than 531.3 ops/s of default Ceph. Arion-60 increases the Ceph data throughput (1.9MB/s) by 58% and reduces the Ceph latency (97.5ms) by 39%. The performance of Ceph-1 is similar to that of Ceph. Fig. 4b illustrates the received and transmitted OSD network traffic normalized by the number of completed operations. Arion-60 reduces the received network load of Ceph —normalized in KB/IO— by 30% and the transmitted by 27%. The bottleneck resource is the server disk I/O caused by synchronous writes.

We examine a metadata-intensive workload with file creations in Figures 4c,d. Arion-60 is comparable to Ceph with respect to performance and load. Ceph-sync achieves higher performance than Arion-60 by 10%, but also increases by 10% the received and transmitted network load. Ceph-sync improves slightly the performance because it handles the metadata updates directly at the MDS instead of fetching them to the client as asynchronous settings do.

We further explore the relative behavior of the two systems using the FIO microbenchmark with Zipfian write pattern of α=1.0001 (e.g., [17]). The benchmark writes a total of 2GB data in a preallocated file of 2GB size with block size in the range 2-16KB. With respect to Ceph (0.9ms) and Ceph-1 (1.5ms) in Fig 5a, Arion-60 achieves lower latency (0.7ms) by 22% and 53%, respectively. We examine the total network traffic received over time at one OSD of each system in Fig 5b. We notice that Ceph terminates at instance 233s with 2.1GB total received traffic. In contrast, Arion-60 ends the experiment at 172s (26% shorter) with received volume 1.2GB (42% lower).

In Figures 5c,d we examine the bandwidth utilization of the journal and filesystem storage device at one of the OSDs. We show Ceph-1 that keeps the durability characteristics similar to those of Arion. The depicted Arion-60 OSD utilizes the journal and filesystem device at 15.0% and 16.4% on average; the respective utilizations of Ceph-1 are 24.4% and 88.4%. We conclude that Arion-60 reduces the filesystem device utilization by 82% with respect to Ceph-1 in the examined case.

Overall, Arion-60 improves the performance of Ceph and Ceph-1 by up to 58%, but also reduces the server network and disk load by up to 42% and 82%, respectively. We experimentally confirmed the improved comparative performance and efficiency of Arion in several other write-intensive workloads (e.g., OLTP, key-value store). We also measured the recovery time of the Arion client in the range 77.4ms-2.622s, depending on the load of client write activity before the crash.

6 Related Work

Filesystems Andrew pioneered client disk-based caching but lacked the explicit separation in data and metadata management of object-based storage [33, 41]. Coda exploited data caching strictly for availability during disconnected operation [18]. During a communication failure, a Coda client logged locally the mutating system calls. At network reconnection, each server received and replayed all the logged operations together as one transaction. On the contrary, Arion continuously logs mutations during normal operation and writes them back to efficiently maintain consistency.

Database consistency can be preserved through transaction correctness [5]. SiloR is a multicore database system that uses logging and checkpointing for fast recovery to a transactionally-consistent state without replication [42]. The Sprite distributed filesystem disabled client caching of files concurrently updated by different clients [27]. Echo introduced ordered write-behind to delay the automatic writing of cached blocks to server disks [23]. NFSv4 delegates request handling to the client for reduced latency and network traffic [29]. Unlike Arion, existing filesystems typically limit client caching to volatile memory without support for durable host-side journaling during normal operation.

Virtualization and cloud VMFS stores disk volumes over shared cluster-based block storage [38]. Capo uses the local disks of the hosts for multicast-based preload and block-based write-through or writeback caching [34]. Ventana combines file-based sharing with the versioning, migration and access control of virtual disks [30]. A client-side manager offers disk-based caching but relies on NFSv3 at the host to connect the virtual machines with object-based storage servers. Therefore, existing storage systems only support block-level caching, or inherit the limited scalability of NFSv3.

CacheFS supports local disk-based caching but is practically limited to read-only filesystems [15]. BlueSky provides on-site NFS-based proxy service of remote cloud storage through local disk caching of journal and log segments [40]. SCFS provides FUSE-based caching of entire files at the client memory and disk without the proxy bottleneck [6]. However, it lacks the journaling integration with a scalable distributed filesystem of Arion for flexible file sharing.

Flash memory Non-volatile memory can be used at the client and server of a distributed filesystem for I/O efficiency [3]. Writeback caching can improve performance, reduce server load, and eliminate cache warmup on restart [2]. Optimistic crash consistency decouples ordering from durability for efficient filesystem consistency [10]. In-place commit over non-volatile memory unifies the buffer cache with journaling [21]. Offering disk-based caching through journaling is an extension of Arion that we plan for future work.

Mercury pointed out the zero recovery point objective (RPO), i.e., no recently-written data lost from a crash. It uses flash memory in the block I/O virtualization stack of the hypervisor to provide write-through caching [9]. Non-zero RPO can be applied for improved performance via block-level writeback caching at the host. Update order is preserved by explicit tracking of the dependency between I/O requests or transaction grouping of modified blocks [19]. Due to concerns about the consistency and durability of these ordering schemes, a recent block-level solution satisfies asynchronously but explicitly the ordering constraints of application-specified write barriers [31]. Nevertheless, host-side block-based caching lacks native support for writable file sharing within or across hosts [9, 19, 31, 2, 16].

7 Discussion

Persistent host-side caching primarily targets the improved performance and efficiency of networked storage. Typically, it uses a block-based interface that inherently lacks both the support for data sharing across different hosts and the ability for interposition in the file-based protocol of a distributed filesystem. It also makes the consistency preservation of network storage a challenging problem because the semantic gap between the file and block interfaces complicates the atomic grouping of dirty blocks by I/O request, and their ordering according to filesystem-imposed dependencies. Finally, the persistence of mapping metadata in block-based caching and the repetitive translation of I/O requests across different storage layers can introduce considerable overheads in networked storage I/O [2, 13].

The original design of Ceph cannot recover any writes that returned after they were only placed at the volatile memory of the client before a crash. Therefore, the Arion architecture is innovative because it adds durability into the client memory cache through journal-based recovery, conditionally propagates the updates to the servers after client reconnection, and also permits the clients to scalably communicate directly with the object servers of the storage backend. Overall, assuming host machines with sufficiently reliable local storage, our approach overcomes several sharing, scalability, and consistency limitations of related existing solutions.

8 Conclusions

For enhanced end-to-end durability of shared storage in the datacenter, we integrate the client of a distributed filesystem with a host-based journal. At the host, we provide local durable storage to dirty data and metadata until they are written to the network servers. We implemented a prototype of the proposed Arion design over the Ceph production distributed filesystem. In a virtualization environment, we experimentally demonstrate promising efficiency and performance results for specific durability levels configured through the frequency of copying dirty blocks to the host-side journal. In our future work we plan to experiment with different types of storage devices; explore interesting tradeoffs among performance, durability and efficiency for demanding applications; and extend the host-based journaling to support caching of blocks evicted from memory.

9 Acknowledgments

We thank our shepherd Jens Teubner for his valuable guidance and the anonymous reviewers for their constructive comments. This research has been supported by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) — Research Funding Program: Thales. Investing in knowledge society through the European Social Fund (project Cloud9).

References

[1] APPUSWAMY, R., LEGTCHENKO, S., AND ROWSTRON, A. Towards paravirtualized network file systems. In *USENIX Workshop on Hot Topics in Storage and File Systems* (Philadelphia, PA, June 2014).

[2] ARTEAGA, D., AND ZHAO, M. Client-side flash caching for cloud systems. In *ACM Intl. Systems and Storage Conf.* (Haifa, Israel, June 2014), pp. 7:1–7:11.

[3] BAKER, M., ASAMI, S., DEPRIT, E., OUSTERHOUT, J., AND SELTZER, M. Non-volatile memory for fast, reliable file systems. In *ACM ASPLOS Conf.* (Boston, MA, Oct. 1992), pp. 10–22.

[4] BARROSO, L., CLIDARAS, J., AND HÖLZLE, U. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2013.

[5] BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Co., Reading, MA, 1987.

[6] BESSANI, A., MENDES, R., OLIVEIRA, T., NEVES, N., CORREIA, M., PASIN, M., AND VERISSIMO, P. SCFS: a shared cloud-backed file system. In *USENIX Annual Technical Conf.* (Philadelphia, PA, 2014), pp. 169–180.

[7] BIRMAN, K., FREEDMAN, D., HUANG, Q., AND DOWELL, P. Overcoming CAP with consistent soft-state replication. *Computer 45*, 2 (Feb. 2012), 50–58.

[8] BOVET, D. P., AND CESATI, M. *Understanding the Linux Kernel*, third ed. O'Reilly Media, Sebastopol, CA, Nov. 2005.

[9] BYAN, S., LENTINI, J., MADAN, A., PABON, L., CONDICT, M., KIMMEL, J., KLEIMAN, S., SMALL, C., AND STORER, M. Mercury: Host-side flash caching for the data center. In *IEEE Intl. Conf. on Mass Storage Systems and Technology* (Pacific Grove, CA, Apr. 2012).

[10] CHIDAMBARAM, V., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Optimistic crash consistency. In *ACM Symp. on Operating Systems Principles* (Farminton, PA, Nov. 2013), pp. 228–243.

[11] CHIDAMBARAM, V., SHARMA, T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Consistency without ordering. In *USENIX Conf. on File and Storage Technologies* (San Jose, CA, Feb. 2012), pp. 73–86.

[12] http://sourceforge.net/projects/filebench/.

[13] HILDERBRAND, D., POVZNER, A., TEWARI, R., AND TARASOV, V. Revisiting the storage stack in virtualized NAS environments. In *USENIX Workshop on I/O Virtualization* (Portland, OR, June 2011).

[14] HOWARD, J. H., KAZAR, M. L., MENEES, S. G., NICHOLS, D. A., SATYANARAYANAN, M., SIDEBOTHAM, R. N., AND WEST, M. J. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems 6*, 1 (Feb. 1988), 51–81.

[15] HOWELLS, D. FS-Cache: a network filesystem caching facility. In *Linux Symposium* (Ottawa, Canada, July 2006).

[16] http://www.fusionio.com/products/ioturbine-virtual.

[17] JOSEPHSON, W. K., BONGO, L. A., FLYNN, D., AND LI, K. DFS: a file system for virtualized flash storage. In *USENIX Conf. on File and Storage Technologies* (San Jose, CA, Feb. 2010), pp. 85–100.

[18] KISTLER, J. J., AND SATYANARAYANAN, M. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems 10*, 1 (Feb. 1992), 3–25.

[19] KOLLER, R., MARMOL, L., RANGASWAMI, R., SUNDARARAMAN, S., TALAGALA, N., AND ZHAO, M. Write policies for host-side flash caches. In *USENIX Conf. on File and Storage Technologies* (San Jose, CA, Feb. 2013), pp. 45–58.

[20] LE, D., HUANG, H., AND WANG, H. Understanding performance implications of nested file systems in a virtualized environment. In *USENIX Conf. File and Storage Technologies* (San Jose, CA, Feb. 2012), pp. 87–100.

[21] LEE, E., BAHN, H., AND NOH, S. H. A unified buffer cache architecture that subsumes journaling functionality via nonvolatile memory. *ACM Transactions on Storage 10*, 1 (Jan. 2014), 1:1–1:17.

[22] LU, L., ZHANG, Y., DO, T., AL-KISWANY, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Physical disentanglement in a container-based file system. In *USENIX Symp. on Operating Systems Design and Implementation* (Broomfield, CO, Oct. 2014).

[23] MANN, T., BIRRELL, A., HISGEN, A., JERIAN, C., AND SWART, G. A coherent distributed file cache with directory write-behind. *ACM Transactions on Computer Systems 12*, 2 (May 1994), 123–164.

[24] MEYER, D. T., AGGARWAL, G., CULLY, B., LEFEBVRE, G., FEELEY, M. J., HUTCHINSON, N. C., AND WARFIELD, A. Parallax: virtual disks for virtual machines. In *ACM European Conf. on Computer Systems* (Glasgow, Scotland, UK, Apr. 2008), pp. 41–54.

[25] MEYER, D. T., WIRES, J., HUTCHINSON, N. C., AND WARFIELD, A. Namespace Management in Virtual Desktops. *USENIX; login: 36*, 1 (Feb. 2011), 6–11.

[26] MICKENS, J., NIGHTINGALE, E. B., ELSON, J., NAREDDY, K., GEHRING, D., FAN, B., KADAV, A., CHIDAMBARAM, V., AND KHAN, O. Blizzard: Fast, cloud-scale block storage for cloud-oblivious applications. In *USENIX Symp. on Networked Systems Design and Implementation* (Seattle, WA, Apr. 2014), pp. 257–273.

[27] NELSON, M. N., WELCH, B. B., AND OUSTERHOUT, J. K. Caching in the Sprite network filesystem. *ACM Transactions on Computer Systems 6*, 1 (Feb. 1988), 134–154.

[28] OPPENHEIMER, D., GANAPATHI, A., AND PATTERSON, D. A. Why do Internet services fail, and what can be done about it? In *USENIX Symp. on Internet Technologies and Systems* (Seattle, WA, Mar. 2003), pp. 1–15.

[29] PAWLOWSKI, B., JUSZCZAK, C., STAUBACH, P., SMITH, C., LEBEL, D., AND HITZ, D. NFS version 3 design and implementation. In *USENIX Summer Technical Conference* (Boston, MA, June 1994), pp. 137–152.

[30] PFAFF, B., GARFINKEL, T., AND ROSENBLUM, M. Virtualization aware file systems: Getting beyond the limitations of virtual disks. In *USENIX Symp. on Networked Systems Design and Implementation* (San Jose, CA, May 2006), pp. 353–366.

[31] QIN, D., BROWN, A. D., AND GOEL, A. Reliable writeback for client-side flash caches. In *USENIX Annual Technical Conf.* (Philadelphia, PA, June 2014), pp. 451–462.

[32] RAJIMWALE, A., CHIDAMBARAM, V., RAMAMURTHI, D., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. Coerced cache eviction and discreet-mode journaling: Dealing with misbehaving disks. In *Intl. Conf. on Dependable Systems and Networks* (Hong Kong, China, June 2011), pp. 518–529.

[33] SATANARAYANAN, M. Scalable, secure, and highly available distributed file access. *Computer 23*, 5 (May 1990), 9–21.

[34] SHAMMA, M., MEYER, D. T., WIRES, J., IVANOVA, M., HUTCHINSON, N. C., AND WARFIELD, A. Capo: Recapitulating storage for virtual desktops. In *USENIX Conf. on File and Storage Technologies* (San Jose, CA, Feb. 2011), pp. 31–45.

[35] TARASOV, V., HILDEBRAND, D., KUENNING, G., AND ZADOK, E. Virtual machine workloads: The case for new benchmarks for NAS. In *USENIX Conf. File and Storage Technologies* (San Jose, CA, Feb. 2013), pp. 307–320.

[36] TARASOV, V., JAIN, D., HILDEBRAND, D., TEWARI, R., KUENNING, G., AND ZADOK, E. Improving I/O performance using virtual disk introspection. In *USENIX Workshop on Hot Topics in Storage and File Systems* (San Jose, CA, June 2013).

[37] THE AUSTIN GROUP. *POSIX.1-2008 Volume 2: System Interfaces*. IEEE Std 1003.1 and The Open Group Base Specifications Issue 7, 2008.

[38] VAGHANI, S. B. Virtual machine file system. *ACM SIGOPS Operating Systems Review 44*, 4 (Dec. 2010), 57–70.

[39] VAN MOOLENBROEK, D. C., APPUSWAMY, R., AND TANENBAUM, A. S. Towards a flexible, lightweight virtualization alternative. In *ACM Intl. Systems and Storage Conf.* (June 2014), pp. 8:1–8:7.

[40] VRABLE, M., SAVAGE, S., AND VOELKER, G. M. Bluesky: A cloud-backed file system for the enterprise. In *USENIX Conf. File and Storage Technologies* (San Jose, CA, Feb. 2012), pp. 237–250.

[41] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In *USENIX Symp. on Operating Systems Design and Implementation* (Seattle, WA, Nov. 2006), pp. 307–320.

[42] ZHENG, W., TU, S., KOHLER, E., AND LISKOV, B. Fast databases with fast durability and recovery through multicore parallelism. In *USENIX Symp. on Operating Systems Design and Implementation* (Broomfield, CO, Oct. 2014), pp. 465–477.

LADS: Optimizing Data Transfers using Layout-Aware Data Scheduling

Youngjae Kim, Scott Atchley, Geoffroy R. Vallée, Galen M. Shipman

Oak Ridge National Laboratory
{kimy1, atchleyes, valleegr, gshipman}@ornl.gov

Abstract

While future terabit networks hold the promise of significantly improving big-data motion among geographically distributed data centers, significant challenges must be overcome even on today's 100 gigabit networks to realize end-to-end performance. Multiple bottlenecks exist along the end-to-end path from source to sink. Data storage infrastructure at both the source and sink and its interplay with the wide-area network are increasingly the bottleneck to achieving high performance. In this paper, we identify the issues that lead to congestion on the path of an end-to-end data transfer in the terabit network environment, and we present a new bulk data movement framework called *LADS* for terabit networks. *LADS* exploits the underlying storage layout at each endpoint to maximize throughput without negatively impacting the performance of shared storage resources for other users. *LADS* also uses the Common Communication Interface (CCI) in lieu of the sockets interface to use zero-copy, OS-bypass hardware when available. It can further improve data transfer performance under congestion on the end systems using buffering at the source using flash storage. With our evaluations, we show that *LADS* can avoid congested storage elements within the shared storage resource, improving I/O bandwidth, and data transfer rates across the high speed networks.

1 Introduction

While "Big Data" is now in vogue, many DOE science facilities have produced a vast amount of experimental and simulation data for many years. Several U.S. Department of Energy (DOE) leadership-computing facilities, such as the Oak Ridge Leadership Computing Facility (OLCF) [23], the Argonne Leadership Computing Facility (ALCF) [1], and the National Energy Research Scientific Computing (NERSC) [21] generate hundreds of petabytes per year of simulation data and are projected to generate in excess of 1 exabyte per year by 2018 [31]. The Big Data and Scientific Discovery report from the DOE, Office of Science, Office of Advanced Scientific Computing Research (ASCR) [5], predicts one of scientific data challenges is the worsening input/output (I/O) bottleneck and the high data movement cost.

To accommodate growing volumes of data, organizations will continue to deploy larger, well provisioned storage infrastructures. These data sets, however, do not exist in isolation. For example, scientists and their collaborators who use the DOE's computational facilities typically have access to additional resources at multiple facilities and/or universities. They use these resources to analyze data generated from experimental facilities or simulation on supercomputers and to validate their results, both of which requires moving the data between geographically dispersed organizations. Some examples of large collaborations include: OLCF petascale simulation needs nuclear interaction datasets processed at NERSC; the ALCF runs a climate simulation and validates the simulation results with climate observation data sets at ORNL data centers.

In order to support the increased growth of data and the desire to move it between organizations, network operators are increasing the capabilities of the network. DOE's Energy Sciences Network (ESnet) [32], for example, has upgraded its network to 100 Gb/s between many DOE facilities, and future deployments will most likely support 400 Gb/s followed by 1 Tb/s throughput. However, these network improvements only contribute to improving the network data transfer rate, not end-to-end data transfer rate from source storage system to sink storage system. The data transfer nodes (DTN) connected to these storage systems and the wide-area network are the focal point for the impedance match between the faster networks and the relatively slower storage systems. In order to improve the scalability, parallel file systems (PFS) use separate servers to service metadata and I/O operations in parallel. To improve I/O throughput, the PFS use ever higher counts of I/O servers connected more disks. DOE sites have widely adopted various PFS to support both high performance I/O and large data sets. Typically, these large scale storage systems use tens to hundreds of I/O servers, each with tens to hundreds of disks, to improve scalability of performance and capacity.

Even as networks reach terabit speeds and PFS grow to exabytes, the storage-to-network mismatch will likely continue to be a major challenge. More importantly, such storage systems are shared resources servicing multiple clients including large computational systems. As contention for these large resources grows, there can be serious Quality-of-Service (QoS) differences between the observed I/O performance by users [11, 38]. Moreover, disk services can degrade while disks in the redundant array of independent disks (RAID) are rebuilding due to failed disks [37, 13]. Also, I/O load imbalance is a serious problem in parallel storage systems [18, 20]. The results showed that a few controllers are highly overloaded while most are not. These observations strongly motivate us to develop a mechanism to avoid temporarily congested servers during data transfers.

We investigate the issues related to designing a data transfer protocol using Common Communication Interface (CCI) [3, 33], that can fully exploit zero-copy, operating system (OS) bypass hardware when available and fall back to sockets when it is not. In particular, we focus on optimizing an end-to-end data transfer, and investigate the interaction between applications, network protocols, and storage systems at both source and sink hosts. We address various design issues for implementing data transfer protocols such as buffer and queue management, synchronization between worker threads, parallelization of remote memory access (RMA) transfers, and I/O optimizations on storage systems. With these design considerations, we develop a **L**ayout-**A**ware **D**ata **S**cheduler (*LADS*).

In this paper, we present *LADS*, a bulk data movement framework for use between PFS which uses the CCI interface for communication. Our primary contribution is that *LADS* uses the *physical view of files*, instead of a logical view. Traditional file transfer tools employ a logical view of files, regardless of how the underlying objects are distributed within the PFS. *LADS*, on the other hand, understands the physical layout of files in which (i) files are composed of data objects, (ii) the set of storage targets that hold the objects, and (iii) the topology of the storage servers and targets.[1] *LADS* aligns all reads and writes to the underlying object size within the PFS. Moreover, *LADS* allows out-of-order object transfers.

Our focus on the objects, rather than on the files, allows us to implement layout-aware I/O scheduling algorithms. With this, we can minimize the stalled I/O times due to congested storage targets by avoiding the congested servers and focusing on idle servers. All other existing data transfer tools [12, 2, 29, 27, 30] implicitly synchronize per file and focus exclusively on the servers that store that one file whether they are busy or not. We also propose a congestion-aware I/O scheduling algorithm, which can increase the data processing rate per thread, leading to a higher data transfer rate. We also implement and evaluate the ideas of hierarchical data transfer using non-volatile memory (NVM) devices. Especially, in an environment where I/O loads on storage dynamically vary, there can be a slow storage target due to congestion.

We conduct a comprehensive evaluation for our proposed ideas using a file size distribution based on a snapshot of one of the file systems of Spider (the previous file system) at ORNL. We compare the performance of our framework with a widely used data transfer program, bbcp [12]. Specifically, in our evaluation with the real file distribution based workload, we observe that our framework yields a 4-5 times higher data transfer rate than bbcp when using eight threads on a node. Also, we find that with a small amount of SSD, *LADS* can improve further the data transfer rate by 37% over a baseline without SSD buffering and far more cost-effectively than provisioning additional DRAM.

2 Background

We first introduce our target environment for DOE data movement frameworks - the data life cycle, network environment, and data storage infrastructure. Next, we define I/O optimization problems at a multi-level hierarchy in the PFS.

2.1 Target Environment

DOE has large HPC systems (e.g. OLCF's Titan and ALCF's Mira) and scientific instruments (e.g. ORNL's SNS and ANL's APS) which generate large, bulk, synchronous I/O. The HPC systems run simulations that have intense computational phases, followed by interprocess communication, and periodically by I/O to checkpoint state or save an output file. The simulation's startup is dominated by a read phase to retrieve the input files as well as the application binary and libraries. The instruments, on the other hand, do not have a read phase and strictly have write workloads that capture measurements. These measurements are triggered by a periodic event such as an accelerated particle hitting a target which generates various energies and sub-particles. The instrument's detectors will capture these events and it must move the data off the device before the next event.

In order to store this data, these systems typically have large PFS connected by a high-performance network.

[1]We use Lustre terminology for object storage servers (OSS) and targets (OST). An OST manages a single device. A single Lustre OSS manages one or more OSTs.

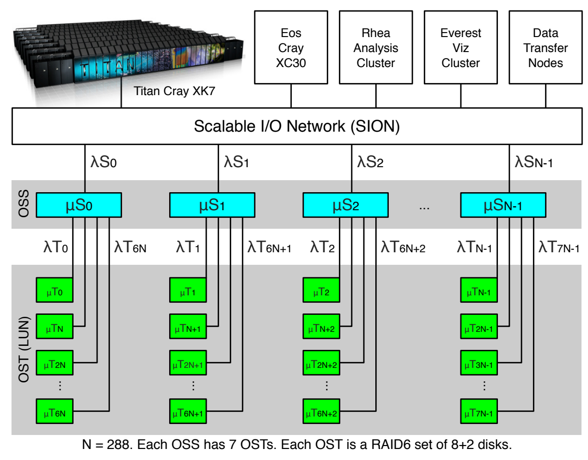


Figure 1: OLCF center-wide PFS and clients

Some sites, such as OLCF, have a center-wide file system accessible by multiple HPC systems as well as by analysis and visualization clusters. In this case, the HPC systems are the primary users and the clusters are secondary users. Given the cost to run these larger resources, one would not want to negatively impact the HPC system's performance due to I/O by the secondary users.

Lastly, most of the scientists using these systems are not located within the facility. Most are from other DOE sites, universities, as well as some commercial entities. They eventually want to move the data back to their institutions in order to further analyze the data. Each site has several data transfer nodes (DTN) that mount the PFS and are connected to DOE's Energy Sciences Network (ESnet). ESnet currently provides 100 Gb/s connectivity to over 40 DOE institutions as well as peering with Internet2 and commercial backbone providers. The DTNs currently have 10 Gb/s NICs but will migrate to 40 Gb/s NICs in the near future. As with the analysis and visualization clusters, use of the DTNs should not negatively impact the I/O of the large HPC systems.

2.2 Spider Storage Systems for Titan

Spider II is OLCF's second generation, center-wide, Lustre system. Its primary client is Titan [25, 19], currently ranked second in the Top500. Titan has 18,688 compute nodes, which mount Spider directly. Titan's I/O traffic passes through 432 I/O nodes, which act as Lustre Networking (LNET) routers between Titan's Cray Gemini network and OLCF's InfiniBand^{TM}(IB) Scalable I/O Network (SION). In addition to Titan, Spider is shared with Eos, a 744 node Cray XC30 system, analysis and visualization clusters, and DTNs. Figure 1 provides an overview of Spider. Currently the file system is accessible via two name-spaces, *atlas1* and *atlas2*, for load-balancing and capacity management purposes. Each namespace has 144 OSSes, which manage seven OSTs each, for a total of 1,008 OSTs per namespace. Each OST represents a RAID-6 set of ten (8+2) disks.

2.3 Problem Definition: I/O Optimization

I/O Contention and Mitigation: A storage server experiences transient congestion when competing I/O requests exceed the capabilities of that server. During these periods, the time to service each new request increases. This is a common occurrence within a PFS when either a large application enters its I/O phase (e.g. writing a checkpoint, reading shared libraries on startup) or multiple applications are accessing files co-located on a subset of OSTs. Disk rebuild processes of a RAID array can also delay I/O services. OS caching and application-level buffering can sometimes mask the congestion for many applications, but data movement tools do not benefit from these techniques. If the congestion occurs on the source side of the transfer, the source's network buffers will drain and eventually stall. On the other hand, congestion at the sink will cause the buffers of both the sink and then the source to fill, eventually stalling the I/O threads at the source. We refer to threads stalled on I/O accesses to congested OSTs as *stalled I/Os*. We try to lower the storage occupancy rate of stalled I/Os in order to minimize the impact of storage congestion on the overall I/O performance using three techniques: *Layout-aware I/O Scheduling*, *OST congestion-aware I/O scheduling*, and *object caching on SSDs*.

Two-level bottlenecks: Figure 1 illustrates the potential places for I/O bottlenecks when accessing OSTs via OSSes in Lustre file systems. For OSS_m, if the arrival rate (λ_{OSSm}) is greater than its service rate (μ_{OSSm}), the server will start to overflow, becoming the bottleneck and its incoming service will be delayed. This can happen if the number of OSTs connected to an OSS is greater than what the network connection to the OSS can handle. To avoid this case, OLCF provisions the number of OSTs per OSS such that $\mu_{OSSm} > \sum_{j=1}^{k} \mu_{OSSn+j}$. Even if λ_{OSSm} is smaller than μ_{OSSm}, OSTs can become the bottleneck. For example, if $\lambda_{OST\,j}$ is greater than $\mu_{OST\,j}$, OST_j becomes the bottleneck. Therefore, *LADS* has to avoid both server and target bottlenecks in a way that it does not assign I/O threads to the overloaded server or target.

Lustre Configuration Impacts I/O Contention: In Lustre, a file's data is stored in a set of objects. The underlying transfers are 1 MB aligned on 1 MB boundaries. If the stripe count is four, then the first object holds offsets 0, 4 MB, 8 MB, etc. Each object is stored on a separate OST. The mapping of the OSTs to the OSSes can impact how a file's object are stored. Figure 2 shows how the OST-to-OSS mapping can physically impact a file's object placement. The default mapping is to assign OSTs sequentially to OSSes. For a file with a stripe count of

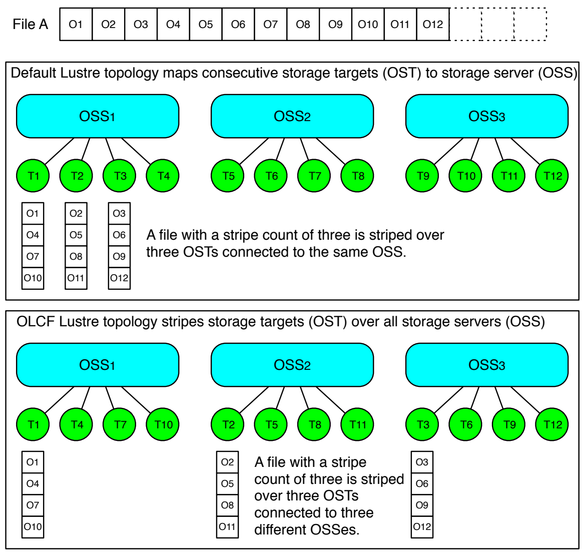

File A
O1 O2 O3 O4 O5 O6 O7 O8 O9 O10 O11 O12
Default Lustre topology maps consecutive storage targets (OST) to storage server (OSS)
OSS1
OSS2
OSS3
A file with a stripe count of three is striped over three OSTs connected to the same OSS.
OLCF Lustre topology stripes storage targets (OST) over all storage servers (OSS)
A file with a stripe count of three is striped over three OSTs connected to three different OSSes.

Figure 2: File striping in Lustre.

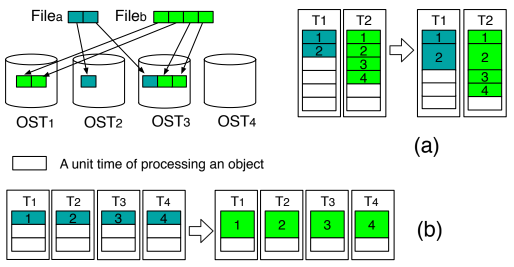


Figure 3: Illustration of slow-down of each job(T) due to OST contention when accessing the same resource at the same time.

three and four OSTs per OSS, the objects will be stored on three OSTs connected to one OSS. OLCF, on the other hand, uses a mapping such that OSTs are assigned round-robin over all the OSSes. In this example, a file with a stripe count of three is assigned to three OSTs and each OST is connected to a separate OSS.

Depending on the choices of storage and networking hardware, the OSS or the OSTs may be the bottleneck. To improve the I/O throughput by minimizing contention, the higher layers need this information.

Logical versus Physical File View: Traditional file transfer tools [12, 2] rely on the logical view of files (called *File-based approach*), which ignores the underlying file system architecture. An I/O thread can be assigned to a complete file, and it should work on the file until the entire file is read or written. If more than one thread is used, these threads might compete for the same OSS or OST, causing server or disk contention respectively. Such contention can result in the slow-down of applications.

To demonstrate how the File-based approach, which is unaware of the underlying file system layout, contributes to the problem of I/O contention in the PFS, we use a simple example in Figure 3, in which we assume each OST can service an object at a time within a fixed service time. In the figure, $File_a$ is striped over OST_2 and OST_3 and $File_b$ is striped over OST_1 and OST_3. In Figure 3(a), Thread 1 (T_1) and T_2 attempt to read $File_a$ and $File_b$ at the same time respectively. T_1 and T_2 read different files, however, T_1 and T_2 can interfere each other on accessing the same OST. Based on a dilation factor model [17], as T_1 and T_2 complete OST_3 , T_1 and T_2 can slow down by 25% and 12.5% respectively. In Figure 3(a), all four threads access different logical regions of the same $File_b$, however, as T_1 and T_2 complete for OST_1, and T_3 and T_4 complete for OST_3. Thus, each thread slows down by 50%. The results of this example indicates that OST contention may increase due the lack of understanding of the physical layout of the file's objects.

In contrast, *LADS*, views the entire workload from a physical point of view based on the underlying file system architecture. *LADS* consider the entire workload of O objects, where O is all of the objects in the N total files, and each object represents one transfer MTU of data. It can also exploit the underlying storage architecture, and can use the file layout information for scheduling accesses of OSTs. Thus it takes into account the S servers and T targets that hold the O objects. We then load-balance based on the physical distribution of the objects. A thread can be assigned to an object of any file on any OST without requiring that all objects of a particular file be transferred before objects of another file.

3 Design of LADS

LADS is motivated to answer a simple question: *how can we exploit the underlying storage architecture to minimize I/O contention at the data source and sink?* In this section, we describe our design rational behind the *LADS* implementation, system architecture, and several key design techniques using a physical view of files on the underlying file system architecture.

We have implemented a data transfer framework with the following main design goals – (i) improved parallelism, (ii) network portability, and (iii) congestion-aware scheduling. Our design tries to maximize parallelism by overlapping as many operations as possible, using use a combined threading and an event-driven model.

3.1 LADS Overview

System Architecture: Figure 4 provides an overview of our design and implementation for I/O sourcing and sinking for a PFS. *LADS* is composed of the follwoing threads: The *Master* thread maintains transfer state, while *I/O* threads read and write objects of files from and to the PFS. The *Comm* thread is in charge of all data transfers between source and sink. In our implementation, there is one Master thread, a configurable num-

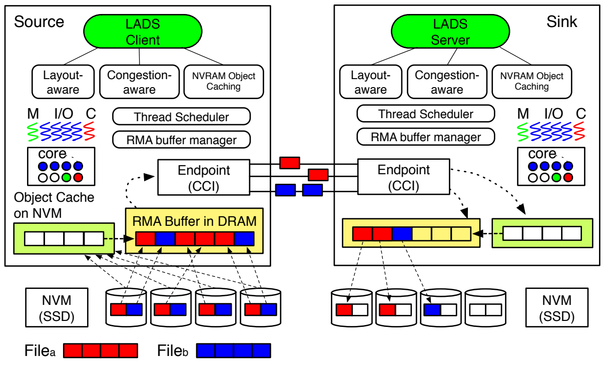


Figure 4: An architecture overview.

ber of I/O threads, and one Comm thread. Because the I/O threads use blocking calls, we allow more threads than cores (i.e. over-subscription). Since we can over-subscribe the cores, the Master and I/O threads block when idle or waiting for a resource. The Comm thread never blocks and always tries to progress communication. The Comm thread generates most of the events that drive the application. If the Comm thread needs a resource (e.g. a buffer) which it cannot get immediately, it will queue the request on the Master's queue and wake the Master.

Several I/O optimization techniques are implemented in *LADS*. In the figure, the layout-aware technique can optimize the unit size of the data accessed by the I/O threads to object size in the underlying file systems, and improved the stalled I/O time when the server is congested. The OST congestion-aware algorithm can avoid the congested servers. NVM can be used as an extended memory region, when the RMA buffer full using the object caching technique. The Comm threads at source and sink, using CCI, pin memory regions for RMA transfers between the Comm threads at source and sink. At the source, if the RMA buffer is full, the Master will notify I/O threads to use the NVM buffer instead of directly copying objects from the PFS into the RMA buffer, thus, it allows pre-loading on the extended memory regions on the NVM. At sink, if the RMA buffer is full, likewise, the extended NVM regions can be used as an intermediate buffer before the PFS to avoid stalling the network transfers.

3.2 Data Structure Overview

We organized the various data structures to minimize false sharing by the various threads. The global state includes a lock which is used to synchronize the threads at startup and shutdown as well as to manage the number of files opened and completed. Other locks are resource specific. There are two wait queues, one for the Master and the other for the I/O threads. When using the SSD to provide additional buffering, it has a wait queue as well.

The Master and I/O thread structures also have a waiter structure that includes their condition variable and an entry for the wait queue. The Master and Comm threads have a work queue implemented using a doubly-linked list. The I/O threads will pull requests off of the OST work queues (described below).

We manage the open files using the GNU tree search interface, which is implemented as a red-black tree. The tree has its own mutex and counter. We manage the RMA and SSD buffers using bitmaps that indicate which offsets are available (the offset is the index in the bitmap multiplied by the object size), an array of contexts (used to store block requests using that buffer), and a mutex. Lastly, because our implementation currently targets Lustre, we have an array of OST pointers. Each OST has a work queue, mutex, queue count, and busy flag. The number of OST queues is determined by the number of OSTs in the PFS. The design can easily be extended to other PFS.

To avoid threads spinning on mutexes as well as "thundering herds" [14] when trying to acquire a resource, we use per-resource wait queues consisting of a linked list, a mutex, and a per-thread condition variable. If the resource is not available, the waiter will acquire the lock, enqueue itself on the resource's wait list, and then block on its own condition variable. When another user wants to release the resource, it acquires the lock, dequeues the first waiter, releases the lock, and signals the waiter's condition variable. This ensures fairness and avoids spinning and thundering herds.

3.3 Object Transfer Protocol

For transferring files, first the source and sink processes (hereafter simply source and sink) need to initialize some state, spawn threads, and exchange some information. The initial state includes a global lock used to synchronize at startup, the various wait queues, the file tree to manage open files, the OST work queues, and the structure for managing access to the RMA buffer. The Master thread initializes its work queue, its wait queue, and the wait queue for I/O threads.

The Comm thread opens a CCI endpoint (send and receive queues, completion queue), allocates its RMA buffer and registers it with CCI, and opens a connection to the remote peer. The source Comm thread sends its maximum object size, number of objects in the RMA buffer, and the memory handle for the RMA buffer. The sink Comm thread accepts the connection request, which triggers the CCI connect event on the source. The I/O threads simply wait for the other threads.

After the CCI initialization step, data transfer will follow these steps at source and sink. The detail figures of thread communication between source and sink hosts can be found in our technical report [16].

Step 1. For each file, (i) the source's master will open the file, determine the file's length and layout (i.e. the size of the stored object and on which OSTs they are located), and generate a `NEW_FILE` request and enqueue that request on the Comm thread's work queue. (ii) The Comm thread generates `NEW_BLOCK` requests for each stored object and enqueue that request on the appropriate OSTs' work queues. (iii) The Comm thread will marshal the `NEW_FILE` request and send it to the sink.

Step 2. At sink, the Comm thread will receive the `NEW_FILE` request and enqueue it on the Master's work queue and wake it up. The Master will open the file, add the file descriptor to the request, change the request type to `FILE_ID` and queue the request on the Comm's work queue. The Comm thread will dequeue it and send it to the source.

Step 3. At source, when the Comm thread receives the `FILE_ID` message, it will wake up *N* I/O threads, where N is the number of OSTs over which the file is striped. An I/O thread first reserves a buffer registered with CCI for RMA. It then determines which OST queue it should access and then dequeues the first `NEW_BLOCK` request. It uses `pread()` to read the data into the RMA buffer. When the read completes, it enqueues the request on the Comm thread's work queue. The Comm thread marshals the request and sends it to the sink. Note, the source's Comm thread's work queue will have intermingled `NEW_FILE` and `NEW_BLOCK` requests thus overlapping file id exchange and block requests.

Step 4. At sink, the Comm thread receives the request and attempts to reserve a RMA buffer. If successful, it initiates a RMA *Read* of the data. If not, it enqueues the request on the Master's work queue and wakes the Master. The Master will sleep on the RMA buffer's wait queue until a buffer is released. It then will queue the request on the Comm's queue, which will then issue the RMA Read.

Step 5. At sink, when the RMA Read completes, it sends a `BLOCK_DONE` message back to the source. The sink's Comm thread determines the appropriate OST by the block's file offset and queues it on the OST's work queue. It then wakes an I/O thread. The I/O thread looks for the next OST to service, dequeues a request, calls `pwrite()` to write the data to disk. When the write completes, it releases the RMA buffer so the Comm thread can initiate another RMA Read.

Step 6. When the source's Comm thread receives the `BLOCK_DONE` message, it releases the RMA buffer and wakes an I/O thread. This pattern continues until all of the file's blocks have been transferred. When all blocks have been written, the source sends a `FILE_DONE` message and closes the file. When the sink receives that message, it too closes the file.

3.4 Scheduling

Layout-aware Scheduling: In a PFS, the file is stored as a collection of objects and stored across multiple servers to improve overall I/O throughput. Best practices for accessing a PFS is for the application to issue large requests in order to reap the benefits of parallel accesses across many servers. A single thread accessing a file will request N objects and can read M objects (assuming $M < N$, and the file is striped over M servers) in parallel at once. If one of the servers is congested, however, the request duration is determined by the slowest server. So the throughput of the request for N objects is determined by the throughput of objects from the congested server. In contract, in our approach, instead of a single thread requesting N objects, we have N threads request one object each from separate servers, because we align all I/O accesses to object boundaries. If one of the requests is delayed by a congested server, the N-1 threads are free to issue new requests to other servers. By the time that the request to the slow server completes, we may be able to retrieve more than N objects.

While the aligned-access technique aims to reduce the I/O stall times and improve overall throughput, it does not specify to which servers to send requests. Most, if not all, data movement tools attempt to move one file at a time (e.g. bbcp, XDD) or a small subset (e.g. GridFTP) at a time. In a PFS, however, a single file is striped over N servers. In the case of the Atlas file system at ORNL, the default is four servers. Although the file system may have hundreds of storage servers, most data movement tools will access a very small subset of them at a time. If one of those servers is congested, overall performance will suffer during the congested period.

Congestion-aware Scheduling: For congestion-aware I/O scheduling, we attempt to avoid intermittently congested storage servers. Given a set of files, we determine where all of the objects reside in the case of reading at the source or determine which servers to stripe the objects over when writing at the sink. We then schedule the accesses based the location of the objects, not based on the file. We enqueue a request for a specific object on a particular OST's queue. The I/O threads then select a queue in a round-robin fashion and dequeue the first request. If another thread is accessing an OST, the other threads skip that queue and move on to the next. If one OST is congested, a thread may stall, but the other threads are free to move on to other, non-congested servers. This is important in a HPC facility like ORNL. The PFS's primary user is the HPC system. We do not want to tune the data movement tools such that they reduce the performance of the HPC system, which is a very expensive resource. Our goal is to maximize performance while using the lightest touch on the PFS.

The basic per-OST queues and simple round-robin scheduling over all the OSTs is able to improve overall I/O performance. We then extend layout-aware scheduling to be congestion-aware by implementing a heuristic algorithm to detect and avoid the congested OSTs. The algorithm can make proactive decisions for selecting storage targets that next I/O threads will work on. The algorithm uses a threshold-based throttling mechanism to further lessen our impact on the HPC system's use of the PFS. When reading at the source, for example, an I/O thread reads a object from its appropriate server and records the read time, and computes an average of multiple object read times during a pre-set time window time (W). If the average read time during W is greater than the pre-set threshold value (T), then it marks the server as congested. The algorithm tells the threads that they should skip congested servers M times. Consequently, the I/O threads avoid the congested servers for a short amount of time, leading to the reduced I/O stall times.

3.5 Object Caching on SSDs

In the case when the sink is experiencing wide-spread congestion (i.e. every I/O thread is accessing a congested server), newly arriving objects will quickly fill the RMA buffer. The sink will then stall the pipeline of RMA Read requests from the source causing the source's RMA buffer to fill. Once full, the source's I/O threads will stall because they have no buffers in which to read. To mitigate this, we investigate using a fast NVM device to extend the buffer space available for reading at the source. Several efforts have introduced new interfaces to efficiently use NVM as an extended memory region [4, 35, 9, 24]. In this work, we specifically use the *NVMalloc* library [35] to build a NVM based, intermediate buffer pool at the source using fast PCIe-based COTS SSDs, where we create a log-file memory-mapped using a `mmap()` system call. The key use of NVM buffer pool is to continue reading objects when the RMA buffer is full at source.

In our implementation, when servicing a new request, an I/O thread tries to reserve a RMA buffer. If one is not available, it attempts to reserve one in the SSD buffer. If successful, it reads into the SSD buffer, enqueues the request on a SSD queue, and wakes the SSD thread. The SSD thread then attempts to acquire a RMA buffer. If not available, it sleeps waiting for a RMA buffer to be released. When a buffer is released, it wakes, reserves the RMA buffer, copies the data to the RMA buffer, and enqueues the request on the Comm thread's work queue. Lastly, the Comm thread marshals the `NEW_BLOCK` and sends it off to the sink.

We could apply the same idea of source-side SSD buffering algorithm for sink-side SSD buffering, however, as we will discuss in the evaluation section in detail, sink-side buffering does little to improve data transfer rates, when buffered I/Os are allowed. Typically writes are buffered I/Os. The key for the SSD buffering is to decide when to use the SSD buffer or not. When using buffered I/Os at sink, our algorithm can not account for the effect of OS's buffer cache and fails to correctly detect congested servers. Using direct I/O for the writes is possible and would allow our algorithm to detect congested servers, but direct I/O performs much worse and we chose not to use it for sink-side SSD buffering.

The copy from SSD buffer to RMA buffer is needed when using hardware that supports zero-copy RMA because the memory must be pinned and registered with the hardware and we cannot register the mapped SSD file. Our design does this even when the hardware does not provide RMA support (i.e. when using sockets underneath CCI). We could detect this scenario and avoid the copy by sending directly from the SSD buffer, but we do not implement at this time. Also, should future interconnects support RMA from NVM, we could avoid the copy as well.

4 Evaluation

For the evaluation of *LADS*, we use two experimental environments, without and with server congestion, and our production environment. First, we show the results of *LADS* without congested servers. We explore the effectiveness of object scheduling in *LADS* versus file based scheduling (e.g., bbcp). We then explore the performance of *LADS* by varying RMA buffer size and scaling performance. Second, we study the impact of server congestion on *LADS* and we propose two mitigating strategies, congestion-aware I/O scheduling and SSD buffering. Lastly, we show the DTN to DTN evaluation with ORNL production systems.

4.1 Experimental Systems

Implementation: *LADS* has been implemented using 4 K lines of C code using Pthreads. We used CCI, which is an open-source network abstraction layer, downloadable from CCI-Forum [6]. The communication model follows a client-server model. On the server side, the *LADS* server daemon has to be run before the *LADS* client starts to transfer data.

Test-bed: In this setup, we used a private testbed with two nodes (source and sink) connected by InfiniBand (IB) QDR (40 Gb/s). The nodes used the IB network to communicate with each other and the disk arrays. We used two Intel® Xeon® CPU E5-2609 @ 2.40 GHz servers with eight cores, 256 GB DRAM, and two node-local Fusion-io Duo SSDs [10] for data transfer nodes (source and sink hosts) running with Linux kernel 2.6.32-358.23.2. Both the source and sink nodes have separate Lustre file systems with one OSS server, one MDS server, and 32 OSTs, mounted over 32 SAS 10K

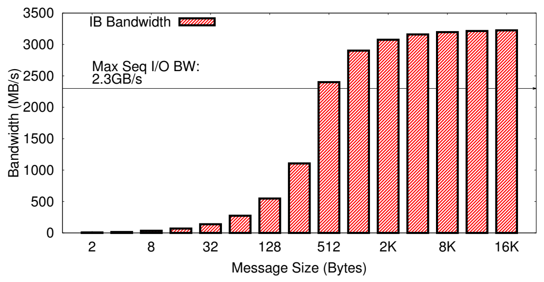


Figure 5: IB vs. storage bandwidth.

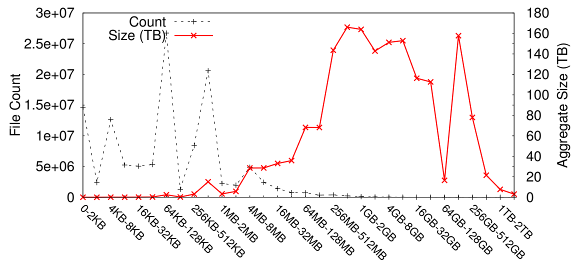


Figure 6: File size distribution.

RPM 1TB drives each. For each file systems, we created 32 logical volume drives on top of the drives to have each disk to become an OST.

To fairly evaluate our implementation framework, we ensured that storage server bandwidth is not over-provisioned with respect to network bandwidth between those source and sink servers (i.e., the network would not be the bottleneck). Figure 5 shows the results on comparing network and storage I/O bandwidths in our test-bed. For storage bandwidth, we ran block level I/O benchmarks [22] on a host to 32 disk volumes in parallel with 1 MB sequential I/O streams on each benchmark with the highest queue depth of 16. The IB bandwidth increases as the message size increases, and it reaches about 3.2 GB/s, whereas the I/O bandwidth is measured around 2.3 GB/s at the most. This testbed allowed us to replicate the temporal congestion of the disks to provide fair comparisons between *LADS* and bbcp.

Production system: We have also tested *LADS* and bbcp between our production Data Transfer Nodes (DTNs), connected to two separate Lustre file systems at ORNL. Each DTN is connected to the OLCF backbone network via a QDR or FDR IB connection to the OLCF's Scalable I/O Network where Atlas' Lustre file systems are mounted (Refer to Figure 1). In our evaluation, we measured the data transfer rate from atlas1 to atlas2 via DTN nodes with *LADS* and bbcp. In order to minimize the OS page-cache effect, we cleared out OS page cache before each measurement at both test-bed and production system.

Workloads: For a realistic performance comparison, we used a file system snapshot taken for a `widow3` partition in the Spider-I file systems hosted by ORNL in 2013 to determine file set sizes. Figure 6 plots a file size distribution in terms of the number of files and the aggregate size of files. We can observe that 90.35% of the files are less than 4 MB and 86.76% are less than 1 MB. Less than 10% of the files are greater than 4 MB. On the other hand, the larger files occupy most of the file system space. For the purpose of our evaluation, we used two representative file sizes to have two file groups; one for *small files* with 10,000 1MB files, and the other for *big files* with 100 1GB files.

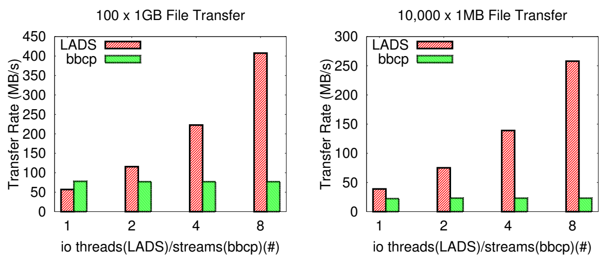
4.2 Scheduling Objects versus Files in an Uncongested Environment

In this section, we show the effectiveness of object scheduling in *LADS* versus file-based scheduling used by bbcp in a controlled, uncongested environment. This section focuses only on the difference between object versus file scheduling; Sections 4.3 and 4.4 will examine two mitigation strategies for congested environments.

Within our controlled test-bed environment, we evaluate the performance of *LADS* for big and small data sets, and compare it against bbcp. In both sets, the stripe count is one (i.e. each file is stored in 1 MB objects on a single OST). We note that our tests with a higher file stripe count are shown in the results of production system in Section 4.5.

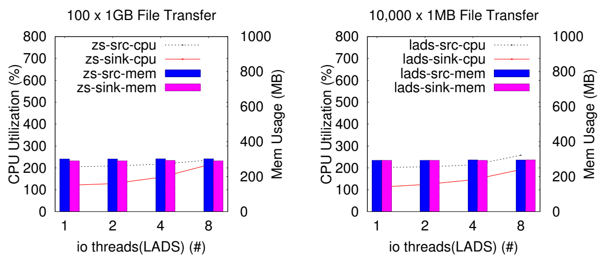
Figures 7 and 8 show the results of *LADS* and bbcp for these workloads. We had multiple runs for each test, however the variability was very small. Both experiments were tested while increasing the number of threads on each application. In *LADS*, we can vary the number of I/O threads, which can maximize CPU utilization on the data transfer node, but use a single Comm thread. On these hosts, *LADS* uses CCI's Verbs transport, which natively uses the underlying InfiniBand interconnect. In bbcp, we can only tune the number of TCP/IP streams for a performance improvement (bbcp always uses a single I/O thread). The streams ran over the same InfiniBand interconnect, but used the IPoIB interface which supports traditional sockets. Using Netperf, we measured IPoIB throughput at almost 1 GB/s. A newer OFED release should provide higher sockets performance, but we ensured that the network was never the bottleneck for these tests. In bbcp, we calculated the TCP window size (W) using the formula for bandwidth-delay product: using ping time (T_{ping}) and a network bandwidth (B_{net}) as follows: $W = T_{ping} \times B_{net}$. We used 10 MB for a TCP window size in our evaluation setup. We have also tested bbcp by varying the block size, however we have seen little performance difference between 1 MB and 4 MB, so we show the results with a block size of 1 MB for bbcp tests.

Performance comparison for object scheduling of *LADS* and file-based scheduling of bbcp: In Fig-



(a) Big Files (b) Small Files

Figure 7: Performance comparisons for *LADS* and bbcp. In bbcp, 10 MB is used for TCP window size.



(a) Big Files (b) Small Files

Figure 8: Resource utilization comparisons for *LADS*. CPU utilization is 800, when all cores are fully utilized.

ure 7(a)(b), we see that *LADS* shows almost a perfect linear scaling in terms of data transfer rate with respect to the increased number of I/O threads, whereas there is little improvement in bbcp with respect to the increased number of TCP/IP streams. bbcp is implemented using a file-based data transfer protocol in which, files are transferred one by one, and multiple TCP streams operate on the same file. Therefore, the bottleneck is determined by how wide the PFS stripes the file. We also found that with bbcp multiple TCP/IP streams will only offer a performance gain when a network speed is moderately slow compared with I/O bandwidth of the storage. Overall, we observe that *LADS* significantly outperforms bbcp for all test cases in Figure 7, except for the results when *LADS* transfer uses one I/O thread for a big file set. In this case, we believe that bbcp is benefiting from hardware-level read-ahead in our testbed. *LADS* did not benefit from it because the round-robin access of the I/O queues might mean that we are accessing an object from a different file the next time we visit this OST and lose the benefit of read-ahead. OLCF production systems disable read-ahead for this reason.

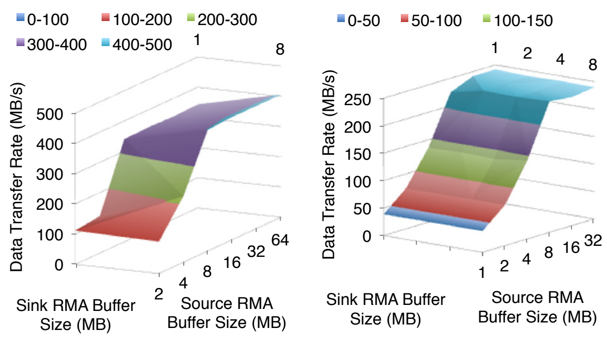
In *LADS*, we observe the maximum throughput at around 400-450MB/s for the experiment of a big data set, which is reasonable based on our test-bed configuration. The block-level throughput for all 16 disks is 2.3GB/s, the file system overhead reduces that by about 40% to 1.3-1.4GB/s. We tested with up to eight threads reducing the optimum to 650-700MB/s. Given thread synchronization overhead, 400-500MB/s is reasonable but improvement is still possible.

Resource utilization in *LADS*: *LADS* uses `DIRECT IO` for the source's read operations to minimize the resource utilization for CPU and memory, while the sink writes using buffered I/O. As we see from Figure 8(a)(b), *LADS* moderately uses system resources, and there is only a slight increase in CPU utilization as the number of I/O threads increases. The more I/O threads involve the more meta data service requests to Lustre file system and more I/O. Overall, *LADS* take advantage of the InfiniBand NIC's offloading abilities and manages well the system resources; CPU utilization stays relatively low even at eight I/O threads. Memory usage never varies and is almost constant at 280-300 MB. We used 256 MB for RMA buffer at source and sink, which accounts for the majority of memory usage.

On the contrary, in bbcp, we observe that CPU and memory usages are very low. For example, for small file workloads, when eight streams are used, their memory usage and CPU utilization are less than 2 MB and 5%, respectively. For big file workloads, with the same configuration, bbcp's memory usage and CPU utilization are at most about 30MB and less than 40%, respectively. Not surprisingly with bbcp's file-based approach, disk I/Os are the bottleneck so that the host resources cannot be fully utilized.

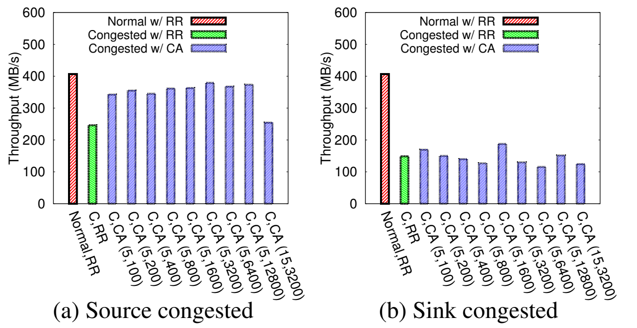
Impact of RMA buffer size in *LADS*: All the experiments in the preceding subsections were done by utilizing a large, fixed amount of DRAM (256 MB) for use as RMA buffers at both the source and sink. Given that DTNs are shared resources and multiple users may be using them concurrently, we want to understand what amount of buffering is necessary.

Figure 9 shows the impact of available RMA buffer sizes at the source and sink on *LADS*. We ran each test multiple times and again the variability was very small. As expected, a larger RMA buffer at the source reduces the waiting time for a slot in the RMA buffer by an I/O thread, which improved data transfer rate from the source. Similarly, a greater size of the RMA buffer at sink can hold more data while I/O threads are busy with writing blocks to OSTs, later reducing the time of an I/O thread has to wait until data are ready to be written from the RMA buffer. Interestingly, with the RMA buffer size increasing, *LADS*'s performance does not always improve. Specifically, we have the following observations: (i) a few RMA buffer slots (a few Megabytes) at sink are sufficient to reach the maximum data transfer rate, and (ii) with the increased RMA buffer at source, *LADS* performance improves. It is because at sink, we allow buffered I/Os, thus writes to disks can be fast, whereas at the source, disk read bandwidth is the bottleneck as



(a) Big File Set (b) Small File Set

Figure 9: Impact of RMA buffer size on *LADS*.



(a) Source congested (b) Sink congested

Figure 10: Comparing average run times of transferring 100 x 1 GB files under normal and congested conditions. Source and sink processes are run with eight I/O threads.

DIRECT I/Os are used for data read on the storage at source. It would be beneficial to let multiple I/O threads read data blocks in parallel into the RMA buffer using multiple slots in the RMA buffer. Therefore, it would be more beneficial to add more RMA slots at the source to improve the data read performance than to increase the RMA buffer size at the sink. We also observe that from the big data set test, a smaller RMA buffer size at sink can be the bottleneck, which never happens in the small data set test. We suspect it may be due to the fact that the small files have a `close()` call after each object is transferred which requires a round-trip to the meta-data server, but we did not investigate further.

We also studied scaling performance of *LADS* and bbcp by increasing the number of source-sink instances. We observed that *LADS* outperforms bbcp in aggregate throughput as the number of paired-instances increases. The detail results can be found in our technical report [16].

4.3 Congestion-aware I/O Scheduling in Congested Environment

In the previous section, we showed the effectiveness of object scheduling compared to file-based scheduling. In this section, we show the effectiveness of a congestion-aware scheduling algorithm on top of object scheduling in *LADS* for variable I/O load environment on storage systems.

Figure 10 shows the run time comparison results of transferring a total of 100 GB of data in both a normal and storage-congested environment. We executed multiple runs for each test, however there was very little variability in measurement between runs. In the figure, “Normal” indicates when there are no congested disks, “C” means a condition where there are congested disks, and “RR” and “CA” represent *Round Robin* and *Congestion-Aware* scheduling algorithms respectively. In (A, B), A means a threshold to determine if disks are congested, and B denotes a number of times the I/O threads skip one or more disks. To simulate congestion, we used a Linux I/O load generator which uses `libaio` [22]. It generates sequential read requests to four disks with an iteration of five seconds, issuing enough requests to generate 310-350 MB/s of I/O. It runs 10 iterations before it moves on to the next four disks. We had the I/O load generator issue 4 MB requests with a queue depth of four.

For Figure 10(a), we tested various parameter settings, to see the effectiveness of our CA algorithm when the source storage is partially in congestion. Overall, we see that the CA performance can improve by 35% over the RR performance when experiencing congestion. The ranges of a performance improvement can be determined in a function of the threshold, and the number of skips over congested servers. We notice that if the threshold value is set too large or if the number of skips for congested servers to be set either too small or too large, the algorithm likely makes false-positive decisions, negating the performance gain from avoiding congested disks.

For Figure 10(b) shows the results for congestion at the sink PFS. Overall, the performance impact is much significantly higher than when source servers are congested. Surprisingly, the congestion-aware scheduling is almost never improving performance, showing execution times as high as those obtained with the RR algorithm. Irrespective of tuning parameter values, the run times are quite random, mainly because our scheduling algorithm failed to detect congested servers. The congestion-aware algorithm measures I/O service time for each object, but our use of buffered I/Os prevented it from accurately measuring the OSTs’ actual level of congestion. We confirmed from our evaluation that most of predictions were false positives, often wrongly assigning I/O threads to busy or overloaded OSTs.

We measured the throughput of bbcp for a congested condition in the storage. The results are shown in Table 1 to compare against the results of *LADS*. We executed multiple runs for each test, however there were very little variability in measurement between runs. The same test-scenarios is used for the *LADS* evaluation presented in Figure 10. It is not surprising that *LADS* is faster than bbcp in both normal and congested conditions. Interestingly, we note that the bbcp run times when the sink is congested are not much different from those under normal conditions, which is most likely due to combination of the OS buffer cache and bbcp's slower communication throughput. It is obvious that buffered I/Os for writes should have been able to hide disk write latency. On the other hand, we observe that bbcp's run time, when the source is experiencing congestion, can increase by 19% over when normal condition. Moreover, bbcp's use of sockets incurs additional copies, user-to-kernel context switches, as well as TCP/IP stack processing. The slower network throughput masks the sink disk congestion. *LADS* clearly benefits from utilizing zero-copy networks when available.

bbcp	Uncongested Condition	Congested (Side)	
		Source	Sink
Runtime	21m53s	26m11s	21m54s
Throughput (MB/s)	78	65	78

Table 1: Run times and throughput for bbcp under normal and congested environment.

4.4 Source-based Buffering using Flash in Congested Environment

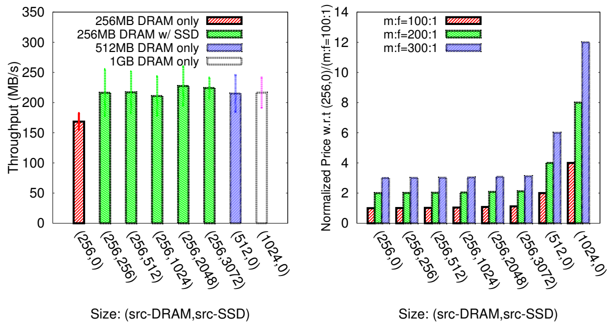
In the previous subsection, we observed that *LADS*' data transfer throughput significantly drops when the sink is overloaded. In this case, the source's RMA buffer becomes full, which stalls the I/O threads from reading additional objects. Therefore, we propose a source-based buffering technique that uses flash-based storage. This source-based SSD buffering utilizes available buffers on flash, which are slower than DRAM yet faster than HDD, to load ahead data blocks to be transferred.

To evaluate it, we slightly modified the overloading workload as we used for Figure 10(b) by inserting ten seconds of idleness between storage congestion periods. During this congestion-free period at sink, source can copy the buffered data from SSD buffer to network RMA buffer. For a fair evaluation, the sink host is set to use only 256 MB RMA buffer, and source and sink run eight I/O threads. The source and sink do not employ the congestion-aware algorithm.

Figure 11(a) shows the results of the effectiveness of the source-based buffering technique using flash. We observe that throughput increases as the available memory for communications at the source increases. However, referring to Figure 11(b), doubling the size of DRAM is very expensive and the same throughput could be achieved using cheaper flash memory.

4.5 Production Environment

DTN to DTN evaluations at ORNL: To evaluate large-



(a) Throughput (b) Price

Figure 11: Performance analysis of SSD-based object buffering at source. In (a), we showed average throughput with 95% confidence intervals in error bars. In (b), $m:f$ denotes the price ratio between DRAM and Flash.

scale performance, we compare the times for transferring a big data set from atlas1 to atlas2 via two DTNs available at ORNL using both *LADS* and bbcp. For this experiment, both bbcp and *LADS* use sockets (in the context of *LADS*, CCI is setup to use its TCP transport) over IPoIB between the source and sink DTNs. The overhead of CCI implementation is quite minimal [3] in which CCI added 150-450 ns to small message latency and no perceptible impact on throughput. On the Lustre Atlas file systems, 1 MB stripe size and a stripe count of four are the default. We ran the experiments twice for every test and Table 2 shows the average throughput (in MB/s). We also want to remind that the Atlas file systems do not use SSDs for buffering.

Threads (#)	1	2	4	8
LADS	58.71	116.30	228.38	407.02
bbcp	59.91	58.46	57.85	59.49

Table 2: Throughput comparison (MB/s).

Table 2 presents the data transfer times for *LADS* and bbcp by increasing the number of threads. We observe that throughput when using *LADS* increases with respect to the increased number of I/O threads, whereas adding streams does not help bbcp. With eight threads, *LADS* shows 6.8 times higher data transfer rate than bbcp. However, bbcp shows slightly higher in throughput than *LADS* for a single thread. As we observed earlier, bbcp's single I/O thread issues larger reads that Lustre converts to multiple object reads, while *LADS*' single I/O thread will only read a single object at a time. I/O parallelism for bbcp is limited to four, which is a Lustre default file stripe count. On the other hand, *LADS* allows multiple I/O threads to operate on multiple objects from differing files, resulting in multiple threads to work on multiple OSTs simultaneously. Therefore, *LADS* can fully take advantage of the parallelism available from multiple object storage targets.

5 Related Work

Many prior studies have performed on the design and implementation of bulk data movement frameworks [2, 12, 29, 27, 26, 30] and their optimization in wide-area networks [34]. GridFTP [2], provided by Globus toolkit, extends the standard File Transfer Protocol (FTP), and provides high speed, reliable, and secure data transfer. It has a striping feature that enables multi-host to multi-host transfers with each host transferring a subset of files, but does not try to schedule based on the underlying object locations. bbcp [12] is another data transfer utility for moving large files securely, and quickly using multiple streams. It uses a single I/O thread and a file based I/O, and its I/O bandwidth is limited by the stripe width of a file. XDD [29] optimizes the disk I/O performance; enabling file access with direct I/Os and multiple threads for parallelism, and varying file offset ordering to improve I/O access times. These tools are useful for moving large data faster and securely from source host to remote host over the network, but none try to schedule based on the underlying object locations or to detect congested storage targets. Other related work has focused on coupling MPI applications over a terabit network infrastructure [33]. It has investigated a model based on MPI-IO and CCI for transferring large data sets between two MPI applications at different sites. This work does not exploit the underlying file system layouts to improving I/O performance for data transfers either.

Storage contention problems remain a challenge for shared file systems [20, 18, 38, 15, 19]. Reads or writes can be stalled at the file system with overloaded storage targets. The storage target can be busy due to a heavy I/O load by some other applications, or when it is part of a RAID rebuild process. Moreover, I/O server (OSS) can experience bursty I/O requests. Consequently, a longer latency from the storage target can violate the Quality-of-Service (QoS) for file operations [11]. The storage contention can occur even if there is one user application. Multiple threads can implement the program, and one or more threads can share the same storage target, causing contentions. Therefore, the program needs a mechanism for multiple accesses to not compete for the same resource, and needs to be designed in a way to minimize the side-effect created by another user I/O streams. In our prior work [15], we examined the I/O performance of traditional versus layout-aware scheduling. And we addressed a few heuristic algorithms to avoid congested servers. However those algorithms were not fully exploited. On the other hand, in this work, we have fully developed an end-to-end data transfer tool using CCI integrated with layout-awareness algorithms and evaluated them.

Our work differs in several key areas from prior works: (i) We use layout-aware data scheduling to maximize parallelism within the PFS' network paths, servers, and disks. (ii) We focus on the total workload of objects without artificially synchronizing on logical files. (iii) We detect server congestion to minimize our impact on the PFS in order to avoid negatively impacting the performance of the PFS' primary customer, a large HPC system. (iv) We use a modern network abstraction layer, CCI, to take advantage of HPC interconnects to improve throughput.

While our work has focused on I/O optimization for Lustre file systems, one could add support for other parallel file systems such as GPFS [28], and Ceph [36]. *LADS* needs four pieces of information about a given file: object size, stripe width, IDs of servers, and object offsets held by each server. If the parallel file system exposes this information to the user, *LADS* could be implemented for that file system. In Ceph [7] for example, it can return a structure of file system data layout using ioctl with some parameters (e.g., CEPH_IOC_GET_LAYOUT). In Eshel et. al. [8], they describe how pNFS used the layout information of a file in PanFS to perform direct and parallel I/Os.

6 Conclusion

Moving large data sets between geographically distributed organizations is a challenging problem which constrains the ability of researchers to share data. Future terabit networks will help improve the network portion of the data transfer, but not the end-to-end transfer, which sources and sinks the data sets in parallel file systems, due to the impedance mismatch between the faster network and much slower storage system. In this study, we identified multiple bottlenecks that exist along the end-to-end data transfer from source and sink host systems in terabit networks, and we proposed *LADS* to demonstrate techniques that can alleviate some end-to-end bottlenecks while at the same time trying not to negatively impact the use of the PFS by other resources, especially large HPC systems. To minimize the effects of transient congestion within a subset of storage servers, *LADS* implemented three I/O optimization techniques: layout-aware scheduling, congestion-aware scheduling, and object caching using SSDs.

Acknowledgments

We thank the reviewers and our shepherd, Nitin Agrawal, for their constructive comments that have significantly improved the paper. This research is sponsored by the Office of Advanced Scientific Computing Research, U.S. Department of Energy and used resources of the Oak Ridge Leadership Computing Facility, located in the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the Department of Energy under Contract DE-AC05-00OR22725.

References

[1] ALCF. Argonne Leadership Computing Facility. https://www.alcf.anl.gov/.

[2] ALLCOCK, W., BRESNAHAN, J., KETTIMUTHU, R., LINK, M., DUMITRESCU, C., RAICU, I., AND FOSTER, I. The Globus Striped GridFTP Framework and Server. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2005), SC '05, pp. 54–64.

[3] ATCHLEY, S., DILLOW, D., SHIPMAN, G. M., GEOFFRAY, P., SQUYRES, J. M., BOSILCA, G., AND MINNICH, R. The Common Communication Interface (CCI). In *Proceedings of the Hot Interconnects* (2011), pp. 51–60.

[4] BADAM, A., AND PAI, V. S. SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (2011), NSDI'11, pp. 211–224.

[5] BILL HARROD. US Department of Energy Big Data and Scientific Discovery. http://www.exascale.org/bdec/sites/www.exascale.org.bdec/files/talk4-Harrod.pdf.

[6] CCI: Common Communication Interface. http://cci-forum.com//.

[7] Ceph. https://github.com/ceph/.

[8] ESHEL, M., HASKIN, R., HILDEBRAND, D., NAIK, M., SCHMUCK, F., AND TEWARI, R. Panache: A Parallel File System Cache for Global File Access. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies* (2010), FAST '10, pp. 155–168.

[9] ESSEN, B. V., HSIEH, H., AMES, S., AND GOKHALE, M. DI-MMAP: A High Performance Memory-Map Runtime for Data-Intensive Applications. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, Salt Lake City, UT, USA, November 10-16, 2012* (2012), pp. 731–735.

[10] FUSION-IO. Fusion-io ioDrive Duo. http://www.fusionio.com/products/iodrive-duo.

[11] GULATI, A., MERCHANT, A., AND VARMAN, P. J. pClock: An Arrival Curve Based Approach for QoS Guarantees in Shared Storage Systems. In *Proceedings of the 7th ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (2007), SIGMETRICS '07, pp. 13–24.

[12] HANUSHEVSKY, A. BBCP. http://www.slac.stanford.edu/~abh/bbcp/.

[13] HOLLAND, M., AND GIBSON, G. A. Parity Declustering for Continuous Operation in Redundant Disk Arrays. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (1992), ASPLOS V, pp. 23–35.

[14] HONEYMAN, P., LEVER, C., MOLLOY, S., AND PROVOS, N. The Linux Scalability Project. Tech. rep., 1999.

[15] KIM, Y., ATCHLEY, S., VALLÉE, G. R., AND SHIPMAN, G. M. Layout-Aware I/O Scheduling for Terabits Data Movement. In *Proceedings of the IEEE International Conference on Big Data - Workshop on Distributed Storage Systems and Coding for BigData* (2013), IEEE Big Data '13, pp. 44–51.

[16] KIM, Y., ATCHLEY, S., VALLÉE, G. R., AND SHIPMAN, G. M. LADS: Optimizing Data Transfers using Layout-Aware Data Scheduling. Tech. Rep. ORNL/TM-2014/251, Oak Ridge National Laboratory, Oak Ridge, TN, January 2015.

[17] LIM, S.-H., HUH, J.-S., KIM, Y., SHIPMAN, G. M., AND DAS, C. R. D-factor: A Quantitative Model of Application Slow-down in Multi-resource Shared Systems. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems* (2012), SIGMETRICS '12, pp. 271–282.

[18] LIU, Q., PODHORSZKI, N., LOGAN, J., AND KLASKY, S. Runtime I/O Re-Routing + Throttling on HPC Storage. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems* (2013), HotStorage '13.

[19] LIU, Y., GUNASEKARAN, R., MA, X., AND VAZHKUDAI, S. S. Automatic Identification of Application I/O Signatures from Noisy Server-Side Traces. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies* (2014), FAST '14, pp. 213–228.

[20] LOFSTEAD, J., ZHENG, F., LIU, Q., KLASKY, S., OLDFIELD, R., KORDENBROCK, T., SCHWAN, K., AND WOLF, M. Managing Variability in the IO Performance of Petascale Storage Systems. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (2010), SC '10, pp. 1–12.

[21] NERSC. National Energy Research Scientific Computing Cente. https://www.nersc.gov/.

[22] OLCF. I/O Benchmark Suite. https://www.olcf.ornl.gov/center-projects/file-system-projects/.

[23] OLCF. Oak Ridge Leadership Computing Facility. https://www.olcf.ornl.gov/.

[24] OPENNVM. OpenNVM. http://opennvm.github.io/.

[25] ORAL, S., SIMMONS, J., HILL, J., LEVERMAN, D., WANG, F., EZELL, M., MILLER, R., FULLER, D., GUNASEKARAN, R., KIM, Y., GUPTA, S., TIWARI, D., VAZHKUDAI, S. S., ROGERS, J. H., DILLOW, D., SHIPMAN, G. M., AND BLAND, A. S. Best Practices and Lessons Learned from Deploying and Operating Large-scale Data-centric Parallel File Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2014), SC '14, pp. 217–228.

[26] REN, Y., LI, T., YU, D., JIN, S., AND ROBERTAZZI, T. Design and Performance Evaluation of NUMA-aware RDMA-based End-to-end Data Transfer Systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2013), SC '13, pp. 48:1–48:10.

[27] REN, Y., LI, T., YU, D., JIN, S., ROBERTAZZI, T., TIERNEY, B. L., AND POUYOUL, E. Protocols for Wide-area Data-intensive Applications: Design and Performance Issues. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2012), SC '12, pp. 34:1–34:11.

[28] SCHMUCK, F., AND HASKIN, R. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies* (2002), FAST '02.

[29] SETTLEMYER, B., DOBSON, J. M., HODSON, S. W., KUEHN, J. A., POOLE, S. W., AND RUWART, T. M. A Technique for Moving Large Data Sets over High-Performance Long Distance Networks. In *Proceedings of the IEEE Symposium on Massive Storage Systems and Technologies* (2011), MSST '11, pp. 1–6.

[30] SUBRAMONI, H., LAI, P., KETTIMUTHU, R., AND PANDA, D. K. High Performance Data Transfer in Grid Environment Using GridFTP over InfiniBand. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing* (2010), CCGRID '10, pp. 557–564.

[31] TORRELLAS, J. Architectures for Extreme-Scale Computing. *Computer 42*, 11 (Nov. 2009), 28–35.

[32] U.S. DEPARTMENT OF ENERGY, OFFICE OF SCIENCE. Energy Science Network (ESnet). http://www.es.net/.

[33] VALLÉE, G., ATCHLEY, S., KIM, Y., AND SHIPMAN, G. M. End-to-End Data Movement Using MPI-IO Over Routed Terabits Infrastructures. In *Proceedings of the 3rd IEEE/ACM International Workshop on Network-aware Data Management* (2013), NDM '13, pp. 9:1–9:8.

[34] VAZHKUDAI, S., SCHOPF, J. M., AND FOSTER, I. F. Predicting the Performance of Wide Area Data Transfers. In *Proceedings of the IEEE 15th International Parallel and Distributed Processing Symposium* (2001), IPDPS '01.

[35] WANG, C., VAZHKUDAI, S. S., MA, X., MENG, F., KIM, Y., AND ENGELMANN, C. NVMalloc: Exposing an Aggregate SSD Store As a Memory Partition in Extreme-Scale Machines. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium* (2012), IPDPS '12, pp. 957–968.

[36] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. Ceph: A Scalable, High-performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (2006), OSDI '06, pp. 307–320.

[37] WELCH, B., UNANGST, M., ABBASI, Z., GIBSON, G., MUELLER, B., SMALL, J., ZELENKA, J., AND ZHOU, B. Scalable Performance of the Panasas Parallel File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies* (2008), FAST '08, pp. 2:1–2:17.

[38] XIE, B., CHASE, J., DILLOW, D., DROKIN, O., KLASKY, S., ORAL, S., AND PODHORSZKI, N. Characterizing Output Bottlenecks in a Supercomputer. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2012), SC '12, pp. 8:1–8:11.

Having Your Cake and Eating It Too: Jointly Optimal Erasure Codes for I/O, Storage and Network-bandwidth

K. V. Rashmi, Preetum Nakkiran, Jingyan Wang, Nihar B. Shah, Kannan Ramchandran
University of California, Berkeley

Abstract

Erasure codes, such as Reed-Solomon (RS) codes, are increasingly being deployed as an alternative to data-replication for fault tolerance in distributed storage systems. While RS codes provide significant savings in storage space, they can impose a huge burden on the I/O and network resources when reconstructing failed or otherwise unavailable data. A recent class of erasure codes, called minimum-storage-regeneration (MSR) codes, has emerged as a superior alternative to the popular RS codes, in that it minimizes network transfers during reconstruction while also being optimal with respect to storage and reliability. However, existing practical MSR codes do not address the increasingly important problem of I/O overhead incurred during reconstructions, and are, in general, inferior to RS codes in this regard. In this paper, we design erasure codes that are *simultaneously optimal in terms of I/O, storage, and network bandwidth*. Our design builds on top of a class of powerful practical codes, called the product-matrix-MSR codes. Evaluations show that our proposed design results in a significant reduction the number of I/Os consumed during reconstructions (a 5$\times$ reduction for typical parameters), while retaining optimality with respect to storage, reliability, and network bandwidth.

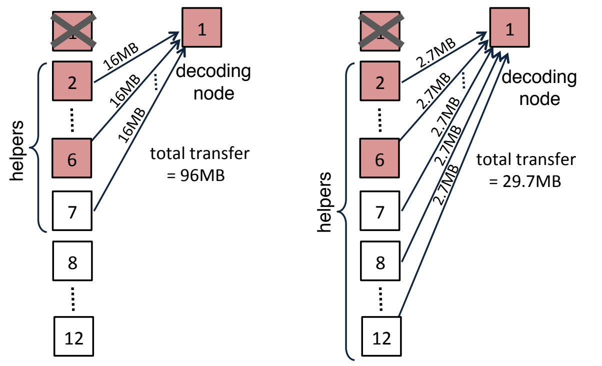
1 Introduction

The amount of data stored in large-scale distributed storage architectures such as the ones employed in data centers is increasing exponentially. These storage systems are expected to store the data in a reliable and available fashion in the face of multitude of temporary and permanent failures that occur in the day-to-day operations of such systems. It has been observed in a number of studies that failure events that render data unavailable occur quite frequently in data centers (for example, see [32, 14, 28] and references therein). Hence, it is imperative that the data is stored in a redundant fashion.

Traditionally, data centers have been employing triple replication in order to ensure that the data is reliable and that it is available to the applications that wish to consume it [15, 35, 10]. However, more recently, the enormous amount of data to be stored has made replication an expensive option. Erasure coding offers an alternative means of introducing redundancy, providing higher levels of reliability as compared to replication while requiring much lower storage overheads [5, 39, 37]. Data centers and cloud storage providers are increasingly turning towards this option [14, 9, 10, 4], with Reed-Solomon (RS) codes [31] being the most popular choice. RS codes make optimal use of storage resources in the system for providing reliability. This property makes RS codes appealing for large-scale, distributed storage systems where storage capacity is one of the critical resources [1]. It has been reported that Facebook has saved multiple Petabytes of storage space by employing RS codes instead of replication in their data warehouse cluster [3].

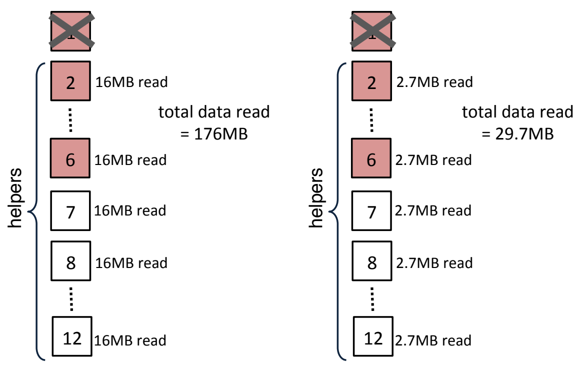
Under RS codes, redundancy is introduced in the following manner: a file to be stored is divided into equal-sized units, which we will call *blocks*. Blocks are grouped into sets of k each, and for each such set of k blocks, r *parity* blocks are computed. The set of these $(k+r)$ blocks consisting of both the data and the parity blocks constitute a *stripe*. The parity blocks possess the property that any k blocks out the $(k+r)$ blocks in a stripe suffice to recover the entire data of the stripe. It follows that the failure of any r blocks in a stripe can be tolerated without any data loss. The data and parity blocks belonging to a stripe are placed on different nodes in the storage network, and these nodes are typically chosen from different racks.

Due to the frequent temporary and permanent failures that occur in data centers, blocks are rendered unavailable from time-to-time. These blocks need to be replaced in order to maintain the desired level of reliability. Under RS codes, since there are no replicas, a missing block



(a) RS (b) MSR (e.g., PM-MSR)

Figure 1: Amount of data transfer involved in reconstruction of block 1 for an RS code with $k = 6$, $r = 6$ and an MSR code with $k = 6$, $r = 6$, $d = 11$, with blocks of size 16MB. The data blocks are shaded.



(a) PM-MSR (b) PM-RBT (this paper)

Figure 2: Amount of data read from disks during reconstruction of block 1 for $k = 6$, $r = 6$, $d = 11$ with blocks of size 16MB.

is replaced by downloading *all* data from (any) k other blocks in the stripe and decoding the desired data from it. We will refer to this operation as a *reconstruction* operation. A reconstruction operation may also be called upon to serve read requests for data that is currently unavailable. Such read requests are called *degraded reads*. Degraded reads are served by reconstructing the requisite data on-the-fly, i.e., immediately as opposed to as a background job.

Let us look at an example. Consider an RS code with $k = 6$ and $r = 6$. While this code has a storage overhead of $2x$, it offers orders of magnitude higher reliability than $3x$ replication. Figure 1a depicts a stripe of this code, and also illustrates the reconstruction of block 1 using the data from blocks 2 to 7. We call the blocks that are called upon during the reconstruction process as *helpers*. In Figure 1a, blocks 2 to 7 are the helpers. In this example, in order to reconstruct a 16MB block, $6 \times 16MB = 96$MB of data is read from disk at the helpers and transferred across the network to the node performing the decoding computations. In general, the disk read and the network transfer overheads during a reconstruction operation is k *times* that under replication. Consequently, under RS codes, reconstruction operations result in a large amount of disk I/O and network transfers, putting a huge burden on these system resources.

There has been considerable interest in the recent past in designing a new class of (network-coding based) codes called *minimum-storage-regenerating (MSR) codes*, which were originally formulated in [12]. Under an MSR code, an unavailable block is reconstructed by downloading a small fraction of the data from any d ($> k$) blocks in the stripe, in a manner that the total amount of data transferred during reconstruction is lower than that in RS codes. Let us consider an example with $k = 6$, $r = 6$ and $d = 11$. For these parameters, reconstruction of block 1 under an MSR code is illustrated in Figure 1b. In this example, the total network transfer is only 29.7MB as opposed to 96MB under RS. MSR codes are optimal with respect to storage and network transfers: the storage capacity and reliability is identical to that under RS codes, while the network transfer is significantly lower than that under RS and in fact, is the minimum possible under any code. However, MSR codes do not optimize with respect to I/Os. The I/O overhead during a reconstruction operation in a system employing an MSR code is, in general, higher than that in a system employing RS code. This is illustrated in Figure 2a which depicts the amount data read from disks for reconstruction of block 1 under a practical construction of MSR codes called product-matrix-MSR (PM-MSR) codes. This entails reading 16MB at each of the 11 helpers, totaling 176MB of data read.

I/Os are a valuable resource in storage systems. With the increasing speeds of newer generation network interconnects and the increasing storage capacities of individual storage devices, I/O is becoming the primary bottleneck in the performance of storage systems. Moreover, many applications that the storage systems serve today are I/O bound, for example, applications that serve a large number of user requests [7] or perform data-intensive computations such as analytics [2]. Motivated by the increasing importance of I/O, in this paper, we investigate practical erasure codes for storage systems that are I/O optimal.

In this paper, we design erasure codes that are *simultaneously optimal in terms of I/O, storage, and network bandwidth* during reconstructions. We first identify two properties that aid in transforming MSR codes to be disk-read optimal during reconstruction while retaining their storage and network optimality. We show that a class of powerful practical constructions for MSR codes, the product-matrix-MSR codes (PM-MSR) [29], indeed satisfy these desired properties. We then present an algorithm to transform *any* MSR code satisfying these properties into a code that is optimal in terms of the amount of data read from disks. We apply our transformation to PM-MSR codes and call the resulting I/O optimal codes as PM-RBT codes. Figure 2b depicts the amount of data read for reconstruction of block 1: PM-RBT entails reading only 2.7MB at each of the 11 helpers, totaling 29.7MB of data read as opposed to 176MB under PM-MSR. We note that the PM-MSR codes operate in the regime $r \geq k-1$, and consequently the PM-RBT codes also operate in this regime.

We implement PM-RBT codes and show through experiments on Amazon EC2 instances that our approach results in 5× reduction in I/Os consumed during reconstruction as compared to the original product-matrix codes, for a typical set of parameters. For general parameters, the number of I/Os consumed would reduce approximately by a factor of $(d-k+1)$. For typical values of d and k, this can result in substantial gains. We then show that if the relative frequencies of reconstruction of blocks are different, then a more holistic system-level design of helper assignments is needed to optimize I/Os across the entire system. Such situations are common: for instance, in a system that deals with degraded reads as well as node failures, the data blocks will be reconstructed more frequently than the parity blocks. We pose this problem as an optimization problem and present an algorithm to obtain the optimal solution to this problem. We evaluate our helper assignment algorithm through simulations using data from experiments on Amazon EC2 instances.

2 Background

2.1 Notation and Terminology

The computations for encoding, decoding, and reconstruction in a code are performed using what is called *finite-field* arithmetic. For simplicity, however, throughout the paper the reader may choose to consider usual arithmetic without any loss in comprehension. We will refer the smallest granularity of the data in the system as a *symbol*. The actual size of a symbol is dependent on the finite-field arithmetic that is employed. For simplicity, the reader may consider a symbol to be a single byte.

A vector will be column vector by default. We will use boldface to denote column vectors, and use T to denote a matrix or vector transpose operation.

We will now introduce some more terminology in addition to that introduced in Section 1. This terminology is illustrated in Figure 3. Let n $(=k+r)$ denote the total number of blocks in a stripe. In order to encode the k data blocks in a stripe, each of the k data blocks are first divided into smaller units consisting of w symbols each. A set of w symbols from each of the k blocks is encoded to obtain the corresponding set of w symbols in the parity blocks. We call the set of data and parities at the granularity of w symbols as a *byte-level* stripe. Thus in a byte-level stripe, $B = kw$ original data symbols (w symbols from each of the k original data blocks) are encoded to generate nw symbols (w symbols for each of the n encoded blocks). The data symbols in different byte-level stripes are encoded independently in an identical fashion. Hence in the rest of the paper, for simplicity of exposition, we will assume that each block consists of a single byte-level stripe. We denote the B original data symbols as $\{m_1, \ldots, m_B\}$. For $i \in \{1, \ldots, n\}$, we denote the w symbols stored in block i by $\{s_{i1}, \ldots, s_{iw}\}$. The value of w is called the *stripe-width*.

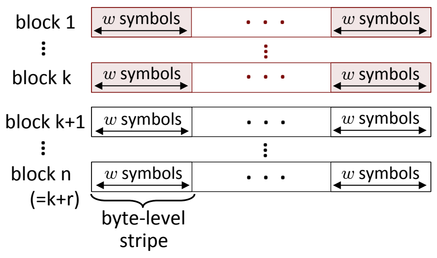


Figure 3: Illustration of notation: hierarchy of symbols, byte-level stripes and block-level stripe. The first k blocks shown shaded are systematic.

During reconstruction of a block, the other blocks from which data is accessed are termed the *helpers* for that reconstruction operation (see Figure 1). The accessed data is transferred to a node that performs the decoding operation on this data in order to recover the desired data. This node is termed the *decoding node*.

2.2 Linear and Systematic Codes

A code is said to be *linear*, if all operations including encoding, decoding and reconstruction can be performed using linear operations over the finite field. Any linear code can be represented using a $(nw \times B)$ matrix G, called its *generator matrix*. The nw encoded symbols can be obtained by multiplying the generator matrix with a vector consisting of the B message symbols:

$$G \left[\begin{array}{cccc} m_1 & m_2 & \cdots & m_B \end{array} \right]^T . \quad (1)$$

We will consider only linear codes in this paper.

In most systems, the codes employed have the property that the original data is available in unencoded (i.e., raw) form in some k of the n blocks. This property is appealing since it allows for read requests to be served directly without having to perform any decoding operations. Codes that possess this property are called *systematic* codes. We will assume without loss of generality that the first k blocks out of the n encoded blocks store the unencoded data. These blocks are called *systematic* blocks. The remaining $(r = n - k)$ blocks store the encoded data and are called *parity* blocks. For a linear systematic code, the generator matrix can be written as

$$\begin{bmatrix} I \\ \hat{G} \end{bmatrix}, \tag{2}$$

where I is a $(B \times B)$ identity matrix, and $\hat{G}$ is a $((nw - B) \times B)$ matrix. The codes presented in this paper are systematic.

2.3 Optimality of Storage Codes

A storage code is said to be optimal with respect to the storage-reliability tradeoff if it offers maximum fault tolerance for the storage overhead consumed. A (k, r) RS code adds r parity blocks, each of the same size as that of the k data blocks. These r parity blocks have the property that any k out of these $(k + r)$ blocks are sufficient to recover all the original data symbols in the stripe. Thus failure of any r arbitrary blocks among the $(k + r)$ blocks in a stripe can be tolerated without any data loss. It is well known from analytical results [22] that this is the maximum possible fault tolerance that can be achieved for the storage overhead used. Hence, RS codes are optimal with respect to the storage-reliability tradeoff.

In [12], the authors introduced another dimension of optimality for storage codes, that of reconstruction bandwidth, by providing a lower bound on the amount of data that needs to be transferred during a reconstruction operation. Codes that meet this lower bound are termed optimal with respect to storage-bandwidth tradeoff.

2.4 Minimum Storage Regenerating Codes

A minimum-storage-regenrating (MSR) code [12] is associated with, in addition to the parameters k and r introduced Section 1, a parameter $d\,(> k)$ that refers to the number of helpers used during a reconstruction operation. For an MSR code, the stripe-width w is dependent on the parameters k and d, and is given by $w = d - k + 1$. Thus, each byte-level stripe stores $B = kw = k(d - k + 1)$ original data symbols.

We now describe the reconstruction process under the framework of MSR codes. A block to be reconstructed can choose *any* d other blocks as helpers from the remaining $(n - 1)$ blocks in the stripe. Each of these d helpers compute some function of the w symbols stored whose resultant is a single symbol (for each byte-level stripe). Note that the symbol computed and transferred by a helper may, in general, depend on the choice of $(d - 1)$ other helpers.

Consider the reconstruction of block f. Let D denote that set of d helpers participating in this reconstruction operation. We denote the symbol that a helper block h transfers to aid in the reconstruction of block f when the set of helpers is denoted by D as t_{hfD}. This resulting symbol is transferred to the decoding node, and the d symbols received from the helpers are used to reconstruct block f.

Like RS codes, MSR codes are optimal with respect to the storage-reliability tradeoff. Furthermore, they meet the lower bound on the amount of data transfer for reconstruction, and hence are optimal with respect to storage-bandwidth tradeoff as well.

2.5 Product-Matrix-MSR Codes

Product-matrix-MSR codes are a class of practical constructions for MSR codes that were proposed in [29]. These codes are linear. We consider the systematic version of these codes where the first k blocks store the data in an unencoded form. We will refer to these codes as *PM-vanilla* codes.

PM-vanilla codes exist for all values of the system parameters k and d satisfying $d \geq (2k - 2)$. In order to ensure that, there are atleast d blocks that can act as helpers during reconstruction of any block, we need atleast $(d + 1)$ blocks in a stripe, i.e., we need $n \geq (d + 1)$. It follows that PM-vanilla codes need a storage overhead of

$$\frac{n}{k} \geq \left(\frac{2k-1}{k}\right) = 2 - \frac{1}{k}. \tag{3}$$

We now briefly describe the reconstruction operation in PM-vanilla codes to the extent that is required for the exposition of this paper. We refer the interested reader to [29] for more details on how encoding and decoding operations are performed. Every block i $(1 \leq i \leq n)$ is assigned a vector $\mathbf{g_i}$ of length w, which we will call the *reconstruction vector* for block i. Let $\mathbf{g_i} = [g_{i1}, \ldots, g_{iw}]^T$. The $n\ (= k + r)$ vectors, $\{\mathbf{g_1}, \ldots, \mathbf{g_n}\}$, are designed such that any w of these n vectors are linearly independent.

During reconstruction of a block, say block f, each of the chosen helpers, take a linear combination of their w stored symbols with the reconstruction vector of the failed block, $\mathbf{g_f}$, and transfer the result to the decoding node. That is, for reconstruction of block f, helper block h computes and transfers the symbol

$$t_{hfD} = \sum_{j=1}^{w} s_{hj} g_{fj}\ , \tag{4}$$

where $\{s_{h1}, \ldots, s_{hw}\}$ are the w symbols stored in block h, and D denotes the set of helpers.

PM-vanilla codes are optimal with respect to the storage-reliability tradeoff and storage-bandwidth tradeoff. However, PM-vanilla codes are not optimal with respect to the amount of data read during reconstruction: The values of most coefficients g_{fj} in the reconstruction vector are non-zero. Since the corresponding *symbol* s_{hj} must be read for every g_{fj} that is non-zero, the absence of sparsity in g_{fj} results in a large I/O overhead during the rebuilding process, as illustrated in Figure 2a (and experimentally evaluated in Figure 7).

3 Optimizing I/O during reconstruction

We will now employ the PM-vanilla codes to construct codes that optimize I/Os during reconstruction, while retaining optimality with respect to storage, reliability and network-bandwidth. In this section, we will optimize the I/Os *locally* in individual blocks, and Section 4 will build on these results to design an algorithm to optimize the I/Os *globally* across the entire system. The resulting codes are termed the *PM-RBT* codes. We note that the methods described here are more broadly applicable to other MSR codes as discussed subsequently.

3.1 Reconstruct-by-transfer

Under an MSR code, during a reconstruction operation, a helper is said to perform *reconstruct-by-transfer* (RBT) if it does not perform any computation and merely transfers one its stored symbols (per byte-level stripe) to the decoding node.[1] In the notation introduced in Section 2, this implies that $\mathbf{g_f}$ in (4) is a unit vector, and

$$t_{hfD} \in \{s_{h1}, \ldots, s_{hw}\} \ .$$

We call such a helper as an *RBT-helper*. At an RBT-helper, the amount of data read from the disks is *equal* to the amount transferred through the network.

During a reconstruction operation, a helper reads requisite data from the disks, computes (if required) the desired function, and transfers the result to the decoding node. It follows that the amount of network transfer performed during reconstruction forms a lower bound on the amount of data read from the disk at the helpers. Thus, a lower bound on the network transfers is also a lower bound on the amount of data read. On the other hand, MSR codes are optimal with respect to network transfers during reconstruction since they meet the associated lower bound [12]. It follows that, under an MSR code, an RBT-helper is optimal with respect to the amount of data read from the disk.

[1]This property was originally titled 'repair-by-transfer' in [34] since the focus of that paper was primarily on node failures. In this paper, we consider more general reconstruction operations that include node-repair, degraded reads etc., and hence the slight change in nomenclature.

We now present our technique for achieving the reconstruct-by-transfer property in MSR codes.

3.2 Achieving Reconstruct-by-transfer

Towards the goal of designing reconstruct-by-transfer codes, we first identify two properties that we would like a helper to satisfy. We will then provide an algorithm to convert any (linear) MSR code satisfying these two properties into one that can perform reconstruct-by-transfer at such a helper.

Property 1*:* The function computed at a helper is independent of the choice of the remaining $(d-1)$ helpers. In other words, for any choice of h and f, t_{hfD} is independent of D (recall the notation t_{hfD} from Section 2.4).

This allows us to simplify the notation by dropping the dependence on D and referring to t_{hfD} simply as t_{hf}.

Property 2*:* Assume Property 1 is satisfied. Then the helper would take $(n-1)$ linear combinations of its own data to transmit for the reconstruction of the other $(n-1)$ blocks in the stripe. We want every w of these $(n-1)$ linear combinations to be linearly independent.

We now show that under the product-matrix-MSR (PM-vanilla) codes, every helper satisfies the two properties enumerated above. Recall from Equation (4), the computation performed at the helpers during reconstruction in PM-vanilla codes. Observe that the right hand side of Equation (4) is independent of 'D', and therefore the data that a helper transfers during a reconstruction operation is dependent only on the identity of the helper and the block being reconstructed. The helper, therefore, does not need to know the identity of the other helpers. It follows that PM-vanilla codes satisfy Property 1.

Let us now investigate Property 2. Recall from Equation (4), the set of $(n-1)$ symbols, $\{t_{h1}, \ldots, t_{h(h-1)}, t_{h(h+1)}, \ldots, t_{hn}\}$, that a helper block h transfers to aid in reconstruction of each the other $(n-1)$ blocks in the stripe. Also, recall that the reconstruction vectors $\{\mathbf{g}_1, \ldots, \mathbf{g}_n\}$ assigned to the n blocks are chosen such that every w of these vectors are linearly independent. It follows that for every block, the $(n-1)$ linear combinations that it computes and transfers for the reconstruction of the other $(n-1)$ blocks in the stripe have the property of any w being independent. PM-vanilla codes thus satisfy Property 2 as well.

PM-vanilla codes are optimal with respect to the storage-bandwidth tradeoff (Section 2.3). However, these codes are not optimized in terms of I/O. As we will show through experiments on the Amazon EC2 instances (Section 5.3), PM-vanilla codes, in fact, have a higher I/O overhead as compared to RS codes. In this section, we will make use of the two properties listed above to transform the PM-vanilla codes into being I/O optimal for reconstruction, while retaining its properties of being storage and network optimal. While we focus on the PM-vanilla codes for concreteness, we remark that the technique described is generic and can be applied to any (linear) MSR code satisfying the two properties listed above.

Under our algorithm, each block will function as an RBT-helper for some w other blocks in the stripe. For the time being, let us assume that for each helper block, the choice of these w blocks is given to us. Under this assumption, Algorithm 1 outlines the procedure to convert the PM-vanilla code (or in general any linear MSR code satisfying the two aforementioned properties) into one in which every block can function as an RBT-helper for w other blocks. Section 4 will subsequently provide an algorithm to make the choice of RBT-helpers for each block to optimize the I/O cost across the entire system.

Let us now analyze Algorithm 1. Observe that each block still stores w symbols and hence Algorithm 1 does not increase the storage requirements. Further, recall from Section 2.5 that the reconstruction vectors $\{\mathbf{g}_{i_{h1}}, \cdots, \mathbf{g}_{i_{hw}}\}$ are linearly independent. Hence the transformation performed in Algorithm 1 is an invertible transformation within each block. Thus the property of being able to recover all the data from any k blocks continues to hold as under PM-vanilla codes, and the transformed code retains the storage-reliability optimality.

Let us now look at the reconstruction process in the transformed code given by Algorithm 1. The symbol transferred by any helper block h for the reconstruction of any block f remains identical to that under the PM-vanilla code, i.e., is as given by the right hand side of Equation (4). Since the transformation performed in Algorithm 1 is invertible within each block, such a reconstruction is always possible and entails the minimum network transfers. Thus, the code retains the storage-bandwidth optimality as well. Observe that for a block f in the set of the w blocks for which a block h intends to function as an RBT-helper, block h now directly stores the symbol $t_{hf} = [s_{h1} \ \cdots \ s_{hw}]\mathbf{g}_f$. As a result, whenever called upon to help block f, block h can directly read and transfer this symbol, thus performing a reconstruct-by-transfer operation. As discussed in Section 3.1, by virtue of its storage-bandwidth optimality and reconstruct-by-transfer, the transformed code from Algorithm 1 (locally) optimizes the amount of data read from disks at the helpers. We will consider optimizing I/O across the entire system in Section 4.

3.3 Making the Code Systematic

Transforming the PM-vanilla code using Algorithm 1 may result in a loss of the systematic property. A further transformation of the code, termed 'symbol remapping', is required to make the transformed code systematic. Symbol remapping [29, Theorem 1] involves transforming the original data symbols $\{m_1, \ldots, m_B\}$ using a bijective transformation before applying Algorithm 1, as described below.

Since every step of Algorithm 1 is linear, the encoding under Algorithm 1 can be represented by a generator matrix, say G_{Alg1}, of dimension $(nw \times B)$ and the encoding can be performed by the matrix-vector multiplication: $G_{Alg1} \left[\ m_1 \ m_2 \ \cdots \ m_B \ \right]^T$. Partition G_{Alg1} as

$$G_{Alg1} = \begin{bmatrix} G_1 \\ G_2 \end{bmatrix}, \qquad (5)$$

where G_1 is a $(B \times B)$ matrix corresponding to the encoded symbols in the first k systematic blocks, and G_2 is an $((nw - B) \times B$ matrix. The symbol remapping step to make the transformed code systematic involves multiplication by G_1^{-1}. The invertibility of G_1 follows from the fact that G_1 corresponds to the encoded symbols in the first k blocks and all the encoded symbols in any set of k blocks are linearly independent. Thus the entire encoding process becomes

$$\begin{bmatrix} G_1 \\ G_2 \end{bmatrix} G_1^{-1} \begin{bmatrix} m_1 \\ m_2 \\ \vdots \\ m_B \end{bmatrix} = \begin{bmatrix} I \\ G_2 G_1^{-1} \end{bmatrix} \begin{bmatrix} m_1 \\ m_2 \\ \vdots \\ m_B \end{bmatrix}, \qquad (6)$$

where I is the $(B \times B)$ identity matrix. We can see that the symbol remapping step followed by Algorithm 1 makes the first k blocks systematic.

Since the transformation involved in the symbol remapping step is invertible and is applied to the data symbols before the encoding process, this step does not affect the performance with respect to storage, reliability, and network and I/O consumption during reconstruction.

3.4 Making Reads Sequential

Optimizing the amount of data read from disks might not directly correspond to optimized I/Os, unless the data read is *sequential*. In the code obtained from Algorithm 1, an RBT-helper reads one symbol per byte-level stripe during a reconstruction operation. Thus, if one chooses the w symbols belonging to a byte-level stripe within a block in a contiguous manner, as in Figure 3, the data read at the helpers during reconstruction operations will be fragmented. In order to the read sequential, we employ the hop-and-couple technique introduced in [27]. The basic idea behind this technique is to choose symbols that are farther apart within a block to form the byte-level stripes. If the stripe-width of the code is w, then choosing symbols that are a $\frac{1}{w}$ fraction of the block size away will make the read at the helpers during reconstruction operations sequential. Note that this technique does not affect the natural sequence of the raw data in the data blocks, so the normal read operations can be served directly without any sorting. In this manner, we ensure

Algorithm 1 Algorithm to achieve reconstruct-by-transfer at helpers

```
**Encode** the data in $k$ data blocks using PM-vanilla code to obtain the $n$ encoded blocks
**for** every block $h$ in the set of $n$ blocks
  **Let** $\{i_{h1}, \ldots, i_{hw}\}$ denote the set of $w$ blocks that block $h$ will help to reconstruct by transfer
  **Let** $\{s_{h1}, \ldots, s_{hw}\}$ denote the set of $w$ symbols that block $h$ stores under $PM-vanilla$
  **Compute** $[s_{h1} \cdots s_{hw}] [\mathbf{g}_{i_{h1}} \cdots \mathbf{g}_{i_{hw}}]$ and store the resulting $w$ symbols in block $h$ instead of the original $w$ symbols
```

that reconstruct-by-transfer optimizes I/O at the helpers along with the amount of data read from the disks.

4 Optimizing RBT-Helper Assignment

In Algorithm 1 presented in Section 3, we assumed the choice of the RBT-helpers for each block to be given to us. Under any such given choice, we saw how to perform a *local* transformation at each block such that reconstruction-by-transfer could be realized under that assignment. In this section, we present an algorithm to make this choice such that the I/Os consumed during reconstruction operations is optimized *globally* across the entire system. Before going into any details, we first make an observation which will motivate our approach.

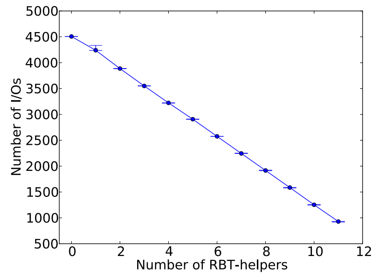
A reconstruction operation may be instantiated in any one of the following two scenarios:

- *Failures:* When a node fails, the reconstruction operation restores the contents of that node in another storage nodes in order to maintain the system reliability and availability. Failures may entail reconstruction operations of either systematic or parity blocks.

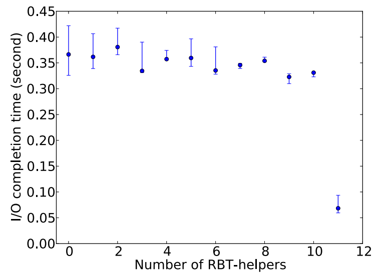
- *Degraded reads:* When a read request arrives, the systematic block storing the requested data may be busy or unavailable. The request is then served by calling upon a reconstruction operation to recover the desired data from the remaining blocks. This is called a degraded read. Degraded reads entail reconstruction of only the systematic blocks.

A system may be required to support either one or both of these scenarios, and as a result, the importance associated to the reconstruction of a systematic block may often be higher than the importance associated to the reconstruction of a parity block.

We now present a simple yet general model that we will use to optimize I/Os holistically across the system. The model has two parameters, δ and p. The relative importance between systematic and parity blocks is captured by the first parameter δ. The parameter δ takes a value between (and including) 0 and 1, and the cost associated with the reconstruction of any parity block is assumed to be δ times the cost associated to the reconstruction of any systematic block. The "cost" can be used to capture the relative difference in the frequency



(a) Total number of I/Os consumed



(b) Maximum of the I/O completion times at helpers

Figure 4: Reconstruction under different number of RBT-helpers for $k = 6$, $d = 11$, and a block size of 16MB.

of reconstruction operations between the parity and systematic blocks or the preferential treatment that one may wish to confer to the reconstruction of systematic blocks in order to serve degraded reads faster.

When reconstruction of any block is to be carried out, either for repairing a possible failure or for a degraded read, not all remaining blocks may be available to help in the reconstruction process. The parameter p $(0 \leq p \leq 1)$ aims to capture this fact: when the reconstruction of a block is to be performed, every other block may individually be unavailable with a probability p independent of all other blocks. Our intention here is to capture the fact that if a block has certain number of helpers that can function as RBT-helpers, not all of them may be available when reconstruction is to be performed.

We performed experiments on Amazon EC2 measuring the number of I/Os performed for reconstruction when precisely j $(0 \leq j \leq d)$ of the available helpers are RBT-helpers and the remaining $(d - j)$ helpers are non-RBT-helpers. The non-RBT-helpers do not perform reconstruct-by-transfer, and are hence optimal with re-

Algorithm 2 Algorithm for optimizing RBT-helper assignment

```
//To compute number of RBT-helpers for each block
Set num_rbt_helpers[block] = 0 for every block
for total_rbt_help = $nw$ to 1
  for block in all blocks
    if num_rbt_helpers[block] < n-1
      Set improvement[block] = Cost(num_rbt_helpers[block]) - Cost(num_rbt_helpers[block]+1)
    else
      Set improvement[block] = -1
  Let max_improvement be the set of blocks with the maximum value of improvement
  Let this_block be a block in max_improvement with the largest value of num_rbt_helpers
  Set num_rbt_helpers[this_block] = num_rbt_helpers[this_block]+1

//To select the RBT-helpers for each block
Call the Kleitman-Wang algorithm [20] to generate a digraph on $n$ vertices with incoming degrees num_rbt_helpers
and all outgoing degrees equal to $w$
for every edge $i \to j$ in the digraph
  Set block $i$ as an RBT-helper to block $j$
```

spect to network transfers but not the I/Os. The result of this experiment aggregated from 20 runs is shown in Figure 4a, where we see that the number of I/Os consumed reduces linearly with an increase in j. We also measured the maximum of the time taken by the d helper blocks to complete the requisite I/O, which is shown Figure 4b. Observe that as long as $j < d$, this time decays very slowly upon increase in j, but reduces by a large value when j crosses d. The reason for this behavior is that the time taken by the non-RBT-helpers to complete the required I/O is similar, but is much larger than the time taken by the RBT-helpers.

Algorithm 2 takes the two parameters δ and p as inputs and assigns RBT-helpers to all the blocks in the stripe. The algorithm optimizes the expected cost of I/O for reconstruction across the system, and furthermore subject to this minimum cost, minimizes the expected time for reconstruction. The algorithm takes a greedy approach in deciding the *number* of RBT-helpers for each block. Observing that, under the code obtained from Algorithm 1, each block can function as an RBT-helper for at most w other blocks, the total RBT-helping capacity in the system is nw. This total capacity is partitioned among the n blocks as follows. The allocation of each unit of RBT-helping capacity is made to the block whose expected reconstruction cost will reduce the most with the help of this additional RBT-helper. The expected reconstruction cost for any block, under a given number of RBT-helper blocks, can be easily computed using the parameter p; the cost of a parity block is further multiplied by δ. Once the number of RBT-helpers for each block is obtained as above, all that remains is to make the choice of the RBT-helpers for each block. In making this choice, the only constraints to be satisfied are the number of RBT-helpers for each block as determined above and that no block can help itself. The Kleitman-Wang algorithm [20] facilitates such a construction.

The following theorem provides rigorous guarantees on the performance of Algorithm 2.

Theorem 1 *For any given (δ, p), Algorithm 2 minimizes the expected amount of disk reads for reconstruction operations in the system. Moreover, among all options resulting in this minimum disk read, the algorithm further chooses the option which minimizes the expected time of reconstruction.*

The proof proceeds by first showing that the expected reconstruction cost of any particular block is convex in the number of RBT-helpers assigned to it. It then employs this convexity, along with the fact that the expected cost must be non-increasing in the number of assigned RBT-helpers, to show that no other assignment algorithm can yield a lower expected cost. We omit the complete proof of the theorem due to space constraints.

The output of Algorithm 2 for $n = 15$, $k = 6$, $d = 11$ and $(\delta = 0.25, p = 0.03)$ is illustrated in Fig 5. Blocks $1, \ldots, 6$ are systematic and the rest are parity blocks. Here, Algorithm 2 assigns 12 RBT-helpers to each of the systematic blocks, and 11 and 7 RBT-helpers to the first and second parity blocks respectively.

The two 'extremities' of the output of Algorithm 2 form two interesting special cases:

1. Systematic (SYS): All blocks function as RBT-helpers for the k systematic blocks.

2. Cyclic (CYC): Block $i \in \{1, \ldots, n\}$ functions as an RBT-helper for blocks $\{i+1, \ldots, i+w\}$ mod n.

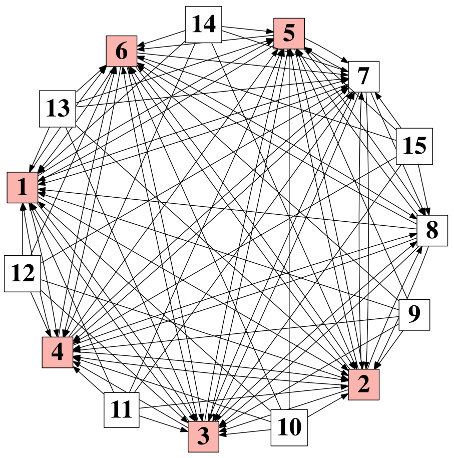


Figure 5: The output of Algorithm 2 for the parameters $n = 15, k = 6, d = 11$, and $(\delta = 0.25, p = 0.03)$ depicting the assignment of RBT-helpers. The directed edges from a block indicate the set of blocks that it helps to reconstruct-by-transfer. Systematic blocks are shaded.

Algorithm 2 will output the SYS pattern if, for example, reconstruction of parity blocks incur negligible cost (δ is close to 0) or if $\delta < 1$ and p is large. Algorithm 2 will output the CYC pattern if, for instance, the systematic and the parity blocks are treated on equal footing (δ is close to 1), or in low churn systems where p is close to 0.

While Theorem 1 provides mathematical guarantees on the performance of Algorithm 2, Section 5.7 will present an evaluation of its performance via simulations using data from Amazon EC2 experiments.

5 Implementation and Evaluation

5.1 Implementation and Evaluation Setting

We have implemented the PM-vanilla codes [29] and the PM-RBT codes (Section 3 and Section 4) in C/C++. In our implementation, we make use of the fact that the PM-vanilla and the PM-RBT codes are both linear. That is, we compute the matrices that represent the encoding and decoding operations under these codes and execute these operations with a single matrix-vector multiplication. We employ the Jerasure2 [25] and GF-Complete [26] libraries for finite-field arithmetic operations also for the RS encoding and decoding operations.

We performed all the evaluations, except those in Section 5.5 and Section 5.6, on Amazon EC2 instances of type m1.medium (with 1 CPU, 3.75 GB memory, and

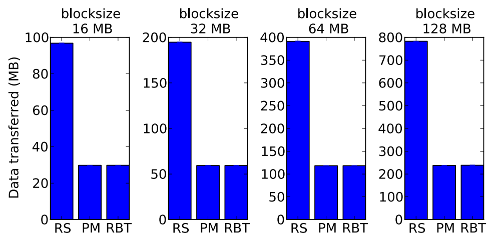


Figure 6: Total amount of data transferred across the network from the helpers during reconstruction. Y-axes scales vary across plots.

attached to 410 GB of hard disk storage). We chose m1.medium type since these instances have hard-disk storage. We evaluated the encoding and decoding performance on instances of type m3.medium which run on an Intel Xeon E5-2670v2 processor with a 2.5GHz clock speed. All evaluations are single-threaded.

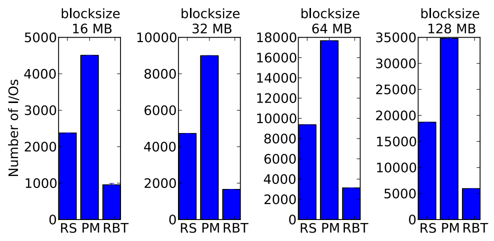
All the plots are from results aggregated over 20 independent runs showing the median values with 25th and 75th percentiles. In the plots, PM refers to PM-vanilla codes and RBT refers to PM-RBT codes. Unless otherwise mentioned, all evaluations on reconstruction are for $(n = 12,\ k = 6,\ d = 11)$, considering reconstruction of block 1 (i.e., the first systematic block) with all $d = 11$ RBT-helpers. We note that all evaluations except the one on decoding performance (Section 5.5) are independent of the identity of the block being reconstructed.

5.2 Data Transfers Across the Network

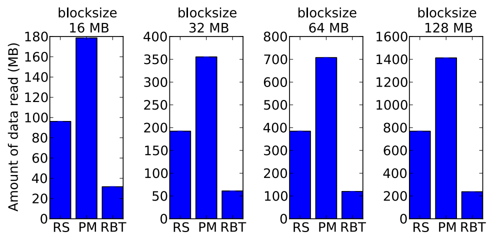
Figure 6 compares the total amount of data transferred from helper blocks to the decoding node during reconstruction of a block. We can see that, both PM-vanilla and PM-RBT codes have identical and significantly lower amount of data transferred across the network as compared to RS codes: the network transfers during the reconstruction for PM-vanilla and PM-RBT are about 4x lower than that under RS codes.

5.3 Data Read and Number of I/Os

A comparison of the total number of disk I/Os and the total amount of data read from disks at helpers during reconstruction are shown in Figure 7a and Figure 7b respectively. We observe that the amount of data read from the disks is as given by the theory across all block sizes that we experimented with. We can see that while PM-vanilla codes provide significant savings in network transfers during reconstruction as compared to RS as seen in Figure 6, they result in an increased number of I/Os. Furthermore, the PM-RBT code leads to a significant reduction in the number of I/Os consumed and the amount of data read from disks during reconstruction.



(a) Total number of disk I/Os consumed



(b) Total amount of data read from disks

Figure 7: Total number of disk I/Os and total amount of data read from disks at the helpers during reconstruction. Y-axes scales vary across plots.

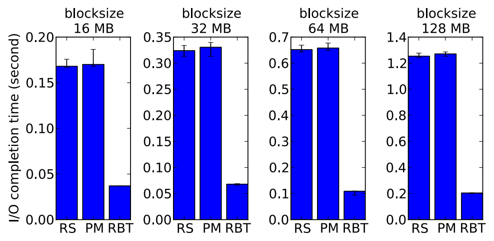


Figure 8: Maximum of the I/O completion times at helpers. Y-axes scales vary across plots.

For all the block sizes considered, we observed approximately a $5x$ reduction in the number of I/Os consumed under the PM-RBT as compared to PM-vanilla (and approximately $3\times$ reduction as compared to RS).

5.4 I/O Completion Time

The I/O completion times during reconstruction are shown in Figure 8. During a reconstruction operation, the I/O requests are issued in parallel to the helpers. Hence we plot the the maximum of the I/O completion times from the $k = 6$ helpers for RS coded blocks and the maximum of the I/O completion times from $d = 11$ helpers for PM-vanilla and PM-RBT coded blocks. We can see that PM-RBT code results in approximately $5\times$ to $6\times$ reduction I/O completion time.

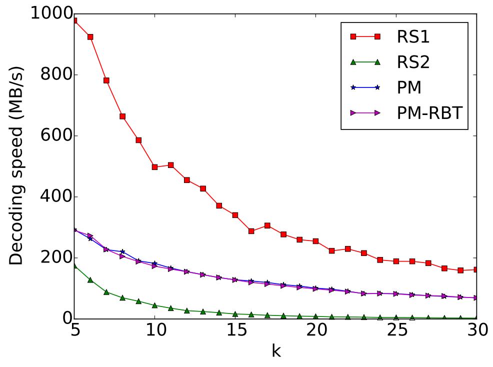


Figure 9: Comparison of decoding speed during reconstruction for various values of k with $n = 2k$, and $d = 2k-1$ for PM-vanilla and PM-RBT.

5.5 Decoding Performance

We measure the decoding performance during reconstruction in terms of the amount of data of the failed/unavailable block that is decoded per unit time. We compare the decoding speed for various values of k, and fix $n = 2k$ and $d = 2k-1$ for both PM-vanilla and PM-RBT. For reconstruction under RS codes, we observed that a higher number of systematic helpers results in a faster decoding process, as expected. In the plots discussed below, we will show two extremes of this spectrum: (RS1) the best case of helper blocks comprising all the existing $(k-1) = 5$ systematic blocks and one parity block, and (RS2) the worst case of helper blocks comprising all the $r = 6$ parity blocks.

Figure 9 shows a comparison of the decoding speed during reconstruction of block 0. We see that the best case (RS1) for RS is the fastest since the operation involves only substitution and solving a small number of linear equations. On the other extreme, the worst case (RS2) for RS is much slower than PM-vanilla and PM-RBT. The actual decoding speed for RS would depend on the number of systematic helpers involved and the performance would lie between the RS1 and RS2 curves. We can also see that the transformations introduced in this paper to optimize I/Os does not affect the decoding performance: PM-vanilla and PM-RBT have roughly the same decoding speed. In our experiments, we also observed that in both PM-vanilla and PM-RBT, the decoding speeds were identical for the n blocks.

5.6 Encoding Performance

We measure the encoding performance in terms of the amount of data encoded per unit time. A comparison of the encoding speed for varying values of k is shown in Figure 10. Here we fix $d = 2k-1$ and $n = 2k$ and vary the values of k. The lower encoding speeds for PM-

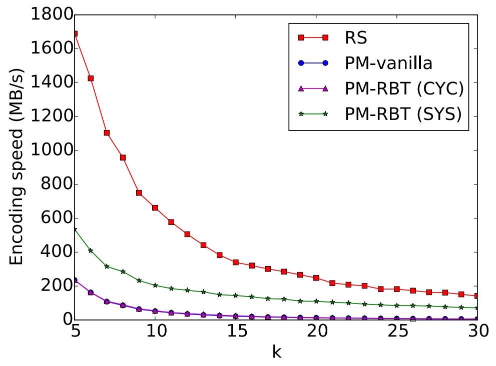


Figure 10: Encoding speed for various values of k with $n = 2k$, and $d = 2k - 1$ for PM-vanilla and PM-RBT.

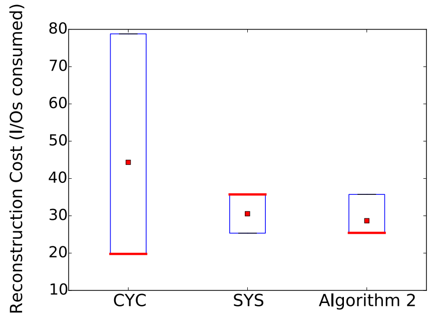


Figure 11: A box plot of the reconstruction cost for different RBT-helper assignments, for $\delta = 0.25$, $p = 0.03$, $n = 15$, $k = 6$, $d = 11$, and block size of 16MB. In each box, the mean is shown by the small (red) square and the median is shown by the thick (red) line.

vanilla and PM-RBT as compared to RS are expected since the encoding complexity of these codes is higher. In RS codes, computing each encoded symbol involves a linear combination of only k data symbols, which incurs a complexity of $O(k)$, whereas in PM-vanilla and PM-RBT codes each encoded symbol is a linear combination of kw symbols which incurs a complexity of $O(k^2)$.

Interestingly, we observe that encoding under PM-RBT with the SYS RBT-helper pattern (Section 4) is significantly faster than that under the PM-vanilla code. This is because the generator matrix of the code under the SYS RBT-helper pattern is sparse (i.e., has many zero-valued entries); this reduces the number of finite-field multiplication operations, that are otherwise computationally heavy. Thus, PM-RBT with SYS RBT-helper pattern results in faster encoding as compared to PM-vanilla codes, in addition to minimizing the disk I/O during reconstruction. Such sparsity does not arise under the CYC RBT-helper pattern, and hence its encoding speed is almost identical to PM-vanilla.

We believe that the significant savings in disk I/O offered by PM-RBT codes outweigh the cost of decreased encoding speed. This is especially true for systems storing immutable data (where encoding is a one-time overhead) and where encoding is performed as a background operation without falling along any critical path. This is true in many cloud storage systems such as Windows Azure and the Hadoop Distributed File System where data is first stored in a triple replicated fashion and then encoded in the background.

Remark: The reader may observe that the speed (MB/s) of encoding in Figure 10 is faster than that of decoding during reconstruction in Figure 9. This is because encoding addresses k blocks at a time while the decoding operation addresses only a single block.

5.7 RBT-helper Assignment Algorithm

As discussed earlier in Section 4, we conducted experiments on EC2 performing reconstruction using different number of RBT-helpers (see Figure 4). We will now evaluate the performance of the helper assignment algorithm, Algorithm 2, via simulations employing the measurements obtained from these experiments. The plots of the simulation results presented here are aggregated from one million runs of the simulation. In each run, we failed one of the n blocks chosen uniformly at random. For its reconstruction operation, the remaining $(n-1)$ blocks were made unavailable (busy) with a probability p each, thereby also making some of the RBT-helpers assigned to this block unavailable. In the situation when only j RBT-helpers are available (for any j in $\{0,\ldots,d\}$), we obtained the cost of reconstruction (in terms of number of I/Os used) by sampling from the experimental values obtained from our EC2 experiments with j RBT-helpers and $(d-j)$ non-RBT-helpers (Figure 4a). The reconstruction cost for parity blocks is weighted by δ.

Figure 11 shows the performance of the RBT-helper assignment algorithm for the parameter values $\delta = 0.25$ and $p = 0.03$. The plot compares the performance of three possible choices of helper assignments: the assignment obtained by Algorithm 2 for the chosen parameters (shown in Figure 5), and the two extremities of Algorithm 2, namely SYS and CYC. We make the following observations from the simulations. In the CYC case, the unweighted costs for reconstruction are homogeneous across systematic and parity blocks due to the homogenity of the CYC pattern, but upon reweighting by δ, the distribution of costs become (highly) bi-modal. In Figure 11, the performance of SYS and the solution obtained from Algorithm 2 are comparable, with the output of Algorithm 2 slightly outperforming SYS. This is as expected since for the given choice of parameter values $\delta = 0.25$ and $p = 0.03$, the output of Algorithm 2 (see Figure 5) is close to SYS pattern.

6 Related Literature

In this section, we review related literature on optimizing erasure-coded storage systems with respect to network transfers and the amount of data read from disks during reconstruction operations.

In [17], the authors build a file system based on the minimum-bandwidth-regenerating (MBR) code constructions of [34]. While system minimizes network transfers and the amount of data read during reconstruction, it mandates additional storage capacity to achieve the same. That is, the system is not optimal with respect to storage-reliability tradeoff (recall from Section 2). The storage systems proposed in [18, 23, 13] employ a class of codes called local-repair codes which optimize the number of blocks accessed during reconstruction. This, in turn, also reduces the amount of disk reads and network transfers. However, these systems also necessitate an increase in storage-space requirements in the form of at least 25% to 50% additional parities. In [21], authors present a system which combines local-repair codes with the graph-based MBR codes presented in [34]. This work also necessitates additional storage space. The goal of the present paper is to optimize I/Os consumed during reconstruction *without* losing the optimality with respect to storage-reliability tradeoff.

[16] and [6], the authors present storage systems based on random network-coding that optimize resources consumed during reconstruction. Here the data that is reconstructed is not identical and is only "functionally equivalent" to the failed data. As a consequence, the system is not systematic, and needs to execute the decoding procedure for serving every read request. The present paper designs codes that are systematic, allowing read requests during the normal mode of operation to be served directly without executing the decoding procedure.

In [27], the authors present a storage system based on a class of codes called Piggybacked-RS codes [30] that also reduces the amount of data read during reconstruction. However, PM-RBT codes provide higher savings as compared to these codes. On the other hand, Piggybacked-RS codes have the advantage of being applicable for all values of k and r, whereas PM-RBT codes are only applicable for $d \geq (2k-2)$ and thereby necessitate a storage overhead of atleast $(2-\frac{1}{k})$. In [19], authors present Rotated-RS codes which also reduce the amount of data read during rebuilding. However, the reduction achieved is significantly lower than that in PM-RBT.

In [33], the authors consider the theory behind reconstruction-by-transfer for MBR codes, which as discussed earlier are not optimal with respect to storage-reliability tradeoff. Some of the techniques employed in the current paper are inspired by the techniques introduced in [33]. In [36] and [38], the authors present optimizations to reduce the amount of data read for reconstruction in array codes with two parities. [19] presents a search-based approach to find reconstruction symbols that optimize I/O for arbitrary binary erasure codes, but this search problem is shown to be NP-hard.

Several works (e.g., [8, 24, 11]) have proposed system-level solutions to reduce network and I/O consumption for reconstruction, such as caching the data read during reconstruction, batching multiple reconstruction operations, and delaying the reconstruction operations. While these solutions consider the erasure code as a black-box, our work optimizes this black-box and can be used in conjunction with these system-level solutions.

7 Conclusion

With rapid increases in the network-interconnect speeds and the advent of high-capacity storage devices, I/O is increasingly becoming the bottleneck in many large-scale distributed storage systems. A family of erasure-codes called minimum-storage-regeneration (MSR) codes has recently been proposed as a superior alternative to the popular Reed-Solomon codes in terms of storage, fault-tolerance and network-bandwidth consumed. However, existing practical MSR codes do not address the critically growing problem of optimizing for I/Os. In this work, we show that it is possible to have your cake and eat it too, in the sense that *we can minimize disk I/O consumed, while simultaneously retaining optimality in terms of both storage, reliability and network-bandwidth.*

Our solution is based on the identification of two key properties of existing MSR codes that can be exploited to make them I/O optimal. We presented an algorithm to transform Product-Matrix-MSR codes into I/O optimal codes (which we term the PM-RBT codes), while retaining their storage and network optimality. Through an extensive set of experiments on Amazon EC2, we have shown that our proposed PM-RBT codes result in significant reduction in the I/O consumed. Additionally, we have presented an optimization framework for helper assignment to attain a system-wide globally optimal solution, and established its performance through simulations based on EC2 experimentation data.

8 Acknowledgements

We thank Ankush Gupta and Diivanand Ramalingam for their contributions to the initial version of the PM-vanilla code implementation. We also thank our shepherd Randal Burns and the anonymous reviewers for their valuable comments.

References

[1] Facebook's Approach to Big Data Storage Challenge. http://www.slideshare.net/Hadoop_Summit/facebooks-approach-to-big-data-storage-challenge.

[2] Hadoop. http://hadoop.apache.org.

[3] HDFS RAID. http://www.slideshare.net/ydn/hdfs-raid-facebook.

[4] Seamless reliability. http://www.cleversafe.com/overview/reliable, Feb. 2014.

[5] ABD-EL-MALEK, M., COURTRIGHT II, W. V., CRANOR, C., GANGER, G. R., HENDRICKS, J., KLOSTERMAN, A. J., MESNIER, M. P., PRASAD, M., SALMON, B., SAMBASIVAN, R. R., SINNAMOHIDEEN, S., STRUNK, J. D., THERESKA, E., WACHS, M., AND WYLIE, J. J. Ursa minor: Versatile cluster-based storage. In *FAST* (2005), vol. 5, pp. 5–5.

[6] ANDRÉ, F., KERMARREC, A.-M., LE MERRER, E., LE SCOUARNEC, N., STRAUB, G., AND VAN KEMPEN, A. Archiving cold data in warehouses with clustered network coding. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), ACM, p. 21.

[7] BEAVER, D., KUMAR, S., LI, H. C., SOBEL, J., VAJGEL, P., ET AL. Finding a needle in haystack: Facebook's photo storage. In *OSDI* (2010), vol. 10, pp. 1–8.

[8] BHAGWAN, R., TATI, K., CHENG, Y. C., SAVAGE, S., AND VOELKER, G. M. Total recall: System support for automated availability management. In *Proc. 1st conference on Symposium on Networked Systems Design and Implementation (NSDI)* (2004).

[9] BORTHAKUR, D. Hdfs and erasure codes (HDFS-RAID). http://hadoopblog.blogspot.com/2009/08/hdfs-and-erasure-codes-hdfs-raid.html, 2009.

[10] CALDER, B., WANG, J., OGUS, A., NILAKANTAN, N., SKJOLSVOLD, A., MCKELVIE, S., XU, Y., SRIVASTAV, S., WU, J., SIMITCI, H., ET AL. Windows Azure storage: a highly available cloud storage service with strong consistency. In *Proc. ACM Symposium on Operating Systems Principles* (2011), pp. 143–157.

[11] CHUN, B.-G., DABEK, F., HAEBERLEN, A., SIT, E., WEATHERSPOON, H., KAASHOEK, M. F., KUBIATOWICZ, J., AND MORRIS, R. Efficient replica maintenance for distributed storage systems. In *NSDI* (2006), vol. 6, pp. 225–264.

[12] DIMAKIS, A. G., GODFREY, P. B., WU, Y., WAINWRIGHT, M., AND RAMCHANDRAN, K. Network coding for distributed storage systems. *IEEE Transactions on Information Theory 56*, 9 (Sept. 2010), 4539–4551.

[13] ESMAILI, K. S., PAMIES-JUAREZ, L., AND DATTA, A. CORE: Cross-object redundancy for efficient data repair in storage systems. In *IEEE International Conference on Big data* (2013), pp. 246–254.

[14] FORD, D., LABELLE, F., POPOVICI, F., STOKELY, M., TRUONG, V., BARROSO, L., GRIMES, C., AND QUINLAN, S. Availability in globally distributed storage systems. In *9th USENIX Symposium on Operating Systems Design and Implementation* (Oct. 2010).

[15] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S. The Google file system. In *ACM SIGOPS Operating Systems Review* (2003), vol. 37, ACM, pp. 29–43.

[16] HU, Y., CHEN, H. C., LEE, P. P., AND TANG, Y. Nccloud: Applying network coding for the storage repair in a cloud-of-clouds. In *USENIX FAST* (2012).

[17] HU, Y., YU, C., LI, Y., LEE, P., AND LUI, J. NCFS: On the practicality and extensibility of a network-coding-based distributed file system. In *International Symposium on Network Coding (NetCod)* (Beijing, July 2011).

[18] HUANG, C., SIMITCI, H., XU, Y., OGUS, A., CALDER, B., GOPALAN, P., LI, J., AND YEKHANIN, S. Erasure coding in Windows Azure Storage. In *USENIX Annual Technical Conference (ATC)* (June 2012).

[19] KHAN, O., BURNS, R., PLANK, J., PIERCE, W., AND HUANG, C. Rethinking erasure codes for cloud file systems: minimizing I/O for recovery and degraded reads. In *Proc. Usenix Conference on File and Storage Technologies (FAST)* (2012).

[20] KLEITMAN, D. J., AND WANG, D.-L. Algorithms for constructing graphs and digraphs with given valences and factors. *Discrete Mathematics 6*, 1 (1973), 79–88.

[21] KRISHNAN, M. N., PRAKASH, N., LALITHA, V., SASIDHARAN, B., KUMAR, P. V., NARAYANAMURTHY, S., KUMAR, R., AND NANDI, S. Evaluation of codes with inherent double replication for hadoop. In *Proc. USENIX HotStorage* (2014).

[22] MACWILLIAMS, F., AND SLOANE, N. *The Theory of Error-Correcting Codes, Part I.* North-Holland Publishing Company, 1977.

[23] MAHESH, S., ASTERIS, M., PAPAILIOPOULOS, D., DIMAKIS, A. G., VADALI, R., CHEN, S., AND BORTHAKUR, D. Xoring elephants: Novel erasure codes for big data. In *VLDB Endowment* (2013).

[24] MICKENS, J., AND NOBLE, B. Exploiting availability prediction in distributed systems. In *NSDI* (2006).

[25] PLANK, J. S., AND GREENAN, K. M. Jerasure: A library in c facilitating erasure coding for storage applications–version 2.0. Tech. rep., Technical Report UT-EECS-14-721, University of Tennessee, 2014.

[26] PLANK, J. S., MILLER, E. L., GREENAN, K. M., ARNOLD, B. A., BURNUM, J. A., DISNEY, A. W., AND MCBRIDE, A. C. Gf-complete: A comprehensive open source library for galois field arithmetic, version 1.0. Tech. Rep. CS-13-716, University of Tennessee, October 2013.

[27] RASHMI, K., SHAH, N. B., GU, D., KUANG, H., BORTHAKUR, D., AND RAMCHANDRAN, K. A hitchhiker's guide to fast and efficient data reconstruction in erasure-coded data centers. In *Proceedings of the 2014 ACM conference on SIGCOMM* (2014), ACM, pp. 331–342.

[28] RASHMI, K. V., SHAH, N. B., GU, D., KUANG, H., BORTHAKUR, D., AND RAMCHANDRAN, K. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster. In *Proc. USENIX HotStorage* (June 2013).

[29] RASHMI, K. V., SHAH, N. B., AND KUMAR, P. V. Optimal exact-regenerating codes for the MSR and MBR points via a product-matrix construction. *IEEE Transactions on Information Theory 57*, 8 (Aug. 2011), 5227–5239.

[30] RASHMI, K. V., SHAH, N. B., AND RAMCHANDRAN, K. A piggybacking design framework for read-and download-efficient distributed storage codes. In *IEEE International Symposium on Information Theory* (July 2013).

[31] REED, I., AND SOLOMON, G. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics 8*, 2 (1960), 300–304.

[32] SCHROEDER, B., AND GIBSON, G. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proc. 5th USENIX conference on File and Storage Technologies (FAST)* (2007).

[33] SHAH, N. B. On minimizing data-read and download for storage-node recovery. *IEEE Communications Letters* (2013).

[34] SHAH, N. B., RASHMI, K. V., KUMAR, P. V., AND RAMCHANDRAN, K. Distributed storage codes with repair-by-transfer and non-achievability of interior points on the storage-bandwidth tradeoff. *IEEE Transactions on Information Theory 58*, 3 (Mar. 2012), 1837–1852.

[35] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The Hadoop distributed file system. In *IEEE Symp. on Mass Storage Systems and Technologies* (2010).

[36] WANG, Z., DIMAKIS, A., AND BRUCK, J. Rebuilding for array codes in distributed storage systems. In *Workshop on the Application of Communication Theory to Emerging Memory Technologies (ACTEMT)* (Dec. 2010).

[37] WEATHERSPOON, H., AND KUBIATOWICZ, J. D. Erasure coding vs. replication: A quantitative comparison. In *International Workshop on Peer-to-Peer Systems* (2002), Springer, pp. 328–337.

[38] XIANG, L., XU, Y., LUI, J., AND CHANG, Q. Optimal recovery of single disk failure in RDP code storage systems. In *ACM SIGMETRICS* (2010), vol. 38, pp. 119–130.

[39] ZHANG, Z., DESHPANDE, A., MA, X., THERESKA, E., AND NARAYANAN, D. Does erasure coding have a role to play in my data center. *Microsoft research MSR-TR-2010 52* (2010).

Efficient MRC Construction with SHARDS

Carl A. Waldspurger Nohhyun Park Alexander Garthwaite Irfan Ahmad
CloudPhysics, Inc.

Abstract

Reuse-distance analysis is a powerful technique for characterizing temporal locality of workloads, often visualized with miss ratio curves (MRCs). Unfortunately, even the most efficient exact implementations are too heavyweight for practical online use in production systems.

We introduce a new approximation algorithm that employs uniform randomized spatial sampling, implemented by tracking references to representative locations selected dynamically based on their hash values. A further refinement runs in constant space by lowering the sampling rate adaptively. Our approach, called *SHARDS* (*S*patially *H*ashed *A*pproximate *R*euse *D*istance *S*ampling), drastically reduces the space and time requirements of reuse-distance analysis, making continuous, online MRC generation practical to embed into production firmware or system software. SHARDS also enables the analysis of long traces that, due to memory constraints, were resistant to such analysis in the past.

We evaluate SHARDS using trace data collected from a commercial I/O caching analytics service. MRCs generated for more than a hundred traces demonstrate high accuracy with very low resource usage. MRCs constructed in a bounded 1 MB footprint, with effective sampling rates significantly lower than 1%, exhibit approximate miss ratio errors averaging less than 0.01. For large traces, this configuration reduces memory usage by a factor of up to 10,800 and run time by a factor of up to 204.

1 Introduction

Caches designed to accelerate data access by exploiting locality are pervasive in modern storage systems. Operating systems and databases maintain in-memory buffer caches containing "hot" blocks considered likely to be reused. Server-side or networked storage caches using flash memory are popular as a cost-effective way to reduce application latency and offload work from rotating disks. Virtually all storage devices — ranging from individual disk drives to large storage arrays — include significant caches composed of RAM or flash memory.

Since cache space consists of relatively fast, expensive storage, it is inherently a scarce resource, and is commonly shared among multiple clients. As a result, optimizing cache allocations is important, and approaches for estimating workload performance as a function of cache size are particularly valuable.

1.1 Cache Utility Curves

Cache utility curves are effective tools for managing cache allocations. Such curves plot a performance metric as a function of cache size. Figure 1 shows an example miss-ratio curve (MRC), which plots the ratio of cache misses to total references for a workload (y-axis) as a function of cache size (x-axis). The higher the miss ratio, the worse the performance; the miss ratio decreases as cache size increases. MRCs come in many shapes and sizes, and represent the historical cache behavior of a particular workload.

Assuming some level of stationarity in the workload pattern at the time scale of interest, its MRC can also be used to predict its future cache performance. An administrator can use a system-wide miss ratio curve to help determine the aggregate amount of cache space to provision for a desired improvement in overall system performance. Similarly, an automated cache manager can utilize separate MRCs for multiple workloads of varying importance, optimizing cache allocations dynamically to achieve service-level objectives.

1.2 Weaker Alternatives

The concept of a *working set* — the set of data accessed during the most recent sample interval [16] — is often used by online allocation algorithms in systems software [12, 54, 61]. While working-set estimation provides valuable information, it doesn't measure data reuse, nor does it predict changes in performance as cache allocations are varied. Without the type of information conveyed in a cache utility curve, administrators or automated systems seeking to optimize cache allocations are forced to resort to simple heuristics, or to engage in trial-and-error tests. Both approaches are problematic.

Heuristics simply don't work well for cache sizing, since they cannot capture the temporal locality profile of a workload. Without knowledge of marginal benefits, for example, doubling (or halving) the cache size for a given workload may change its performance only slightly, or by a dramatic amount.

Trial-and-error tests that vary the size of a cache and measure the effect are not only time-consuming and ex-

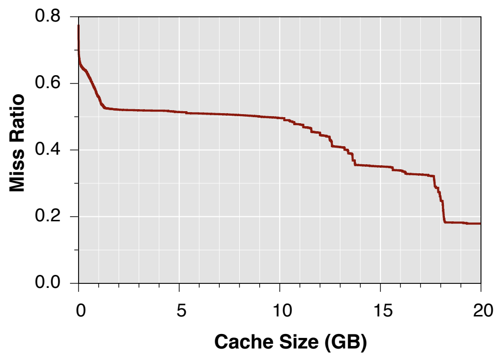


Figure 1: **Example MRC.** A miss ratio curve plots the ratio of cache misses to total references, as a function of cache size.

pensive, but also present significant risk to production systems. Correct sizing requires experimentation across a range of cache allocations; some might induce thrashing and cause a precipitous loss of performance. Long-running experiments required to warm up caches or to observe business cycles may exacerbate the negative effects. In practice, administrators rarely have time for this.

1.3 MRC Construction

Although cache utility curves are extremely useful for planning and optimization, the algorithms used to construct them are computationally expensive. To construct an exact MRC, it is necessary to observe data reuse over the access trace. *Every* accessed location must be tracked and stored in data structures during trace processing, resulting in large overheads in both time and space.

The technique due to Mattson *et al.* [34] scans the trace of references to collect a histogram of reuse distances. The *reuse distance* for an access to a block B is measured as the number of other intervening unique blocks referenced since the previous access to B. The number of times a particular reuse distance occurs is collected while processing the trace, over all possible reuse distances. Conceptually, for modeling LRU, accessed blocks are totally ordered in a stack from most recent to least recent access. On an access to block B, it:

- determines the reuse distance of B as:
  $D =$ stack depth of B (for first access to B, $D = \infty$),
- records D in a reuse-distance histogram, and
- moves B to the top of stack.

Standard implementations maintain a balanced tree to track the most recent references to each block and compute reuse distances efficiently, and employ a hash table for fast lookups into this tree. For a trace of length N containing M unique references, the most efficient implementations of this algorithm have an asymptotic cost of $O(N \log M)$ time and $O(M)$ space.

Given the non-linear computation cost and unbounded memory requirements, it is impractical to perform real-time analysis in production systems. Even when processing can be delayed and performed offline from a trace file, memory requirements may still be excessive.[1] This is especially important when modeling large storage caches; in contrast to RAM-based caches, affordable flash cache capacities often exceed 1 TB, requiring many gigabytes of RAM for traditional MRC construction.

1.4 Our Contributions

We introduce a new approach for reuse-distance analysis that constructs accurate miss ratio curves using only modest computational resources. We call our technique *SHARDS*, for *S*patially *H*ashed *A*pproximate *R*euse *D*istance *S*ampling. It employs randomized spatial sampling, implemented by tracking only references to representative locations, selected dynamically based on a function of their hash values. We further introduce an extended version of SHARDS which runs in *constant* space, by lowering the sampling rate adaptively.

The SHARDS approximation requires several orders of magnitude less space and time than exact methods, and is inexpensive enough for practical online MRC construction in high-performance systems. The dramatic space reductions also enable analysis of long traces that is not feasible with exact methods. Traces that consume many gigabytes of RAM to construct exact MRCs require less than 1 MB for accurate approximations.

This low cost even enables concurrent evaluation of different cache configurations (*e.g.*, block size or write policy) using multiple SHARDS instances. We also present a related generalization to non-LRU policies.

We have implemented SHARDS in the context of a commercial I/O caching analytics service for virtualized environments. Our system streams compressed block I/O traces for VMware virtual disks from customer data centers to a cloud-based backend that constructs approximate MRCs efficiently. A web-based interface reports expected cache benefits, such as the cache size required to reduce average I/O latency by specified amounts. Running this service, we have accumulated a large number of production traces from customer environments.

For this paper, we analyzed both exact and approximate MRCs for more than a hundred virtual disks from our trace library, plus additional publicly-available block I/O traces. Averaged across all traces, the miss ratios of the approximated MRCs, constructed using a 0.1% sampling rate, deviate in absolute value from the exact MRCs by an average of less than 0.02; *i.e.*, the approximate sampled miss ratio is within 2 percentage points of the value calculated exactly using the full trace.

Moreover, approximate MRCs constructed using a fixed sample-set size, with only 8K samples in less than

[1] We have collected several single-VM I/O traces for which conventional MRC construction does not fit in 64 GB RAM.

1 MB memory, deviate by an average of less than 0.01 from the exact full trace values. This high accuracy is achieved despite dramatic memory savings by a factor of up to $10,800\times$ for large traces, and a median of $185\times$ across all traces. The computation cost is also reduced up to $204\times$ for large traces, with a median of $22\times$.

The next section presents the SHARDS algorithm, along with an extended version that runs in constant space. Details of our MRC construction implementation are examined in Section 3. Section 4 evaluates SHARDS through quantitative experiments on more than a hundred real-world I/O traces. Related work is discussed in Section 5. Finally, we summarize our conclusions and highlight opportunities for future work in Section 6.

2 SHARDS Sampling Algorithm

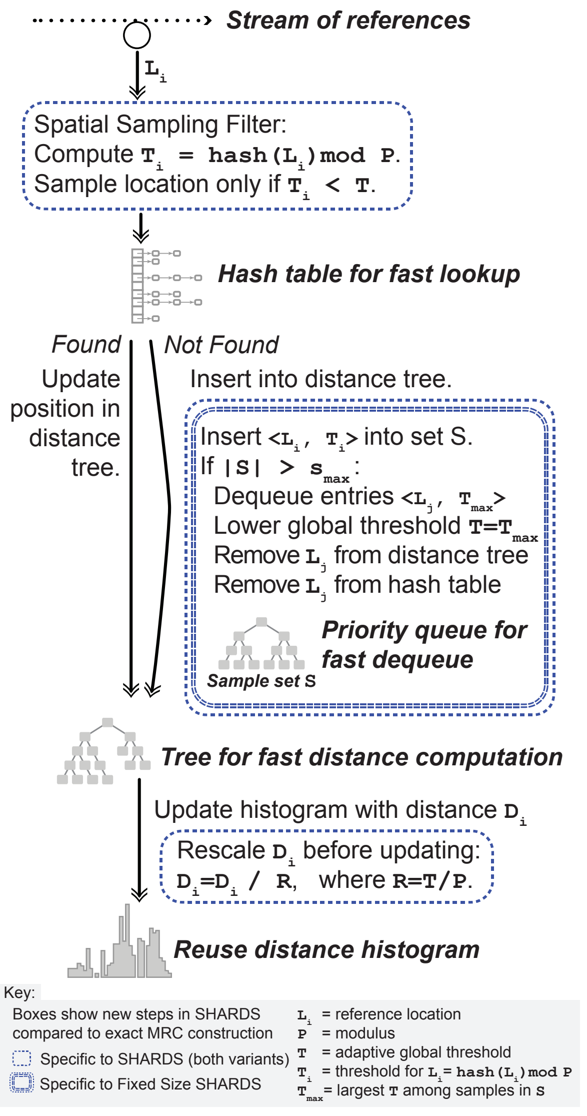
Our core idea is centered around a simple question: what if we compute reuse distances for a randomly sampled subset of the referenced blocks? The answer leads to SHARDS, a new algorithm based on spatially-hashed sampling. Despite the focus on storage MRCs, this approach can be applied more generally to approximate other cache utility curves, with any stream of references containing virtual or physical location identifiers.

2.1 Basic SHARDS

SHARDS is conceptually simple — for each referenced location L, the decision of whether or not to sample L is based on whether $hash(L)$ satisfies some condition. For example, the condition $hash(L) \bmod 100 < K$ samples approximately K percent of the entire location space. Assuming a reasonable hash function, this effectively implements uniform random spatial sampling.

This method has several desirable properties. As required for reuse distance computations, it ensures that all accesses to the same location will be sampled, since they will have the same hash value. It does not require any prior knowledge about the system, its workload, or the location address space. In particular, no information is needed about the set of locations that may be accessed by the workload, nor the distribution of accesses to these locations. As a result, SHARDS sampling is effectively stateless. In contrast, explicitly pre-selecting a random subset of locations may require significant storage, especially if the location address space is large. Often, only a small fraction of this space is accessed by the workload, making such pre-selection especially inefficient.

More generally, using the sampling condition $hash(L) \bmod P < T$, with modulus P and threshold T, the effective sampling rate is $R = T/P$, and each sample represents $1/R$ locations, in a statistical sense. The sampling rate may be varied by changing the threshold



Stream of references
L_i
Spatial Sampling Filter:
Compute T_i = hash(L_i) mod P.
Sample location only if T_i < T.
Hash table for fast lookup
Found
Update position in distance tree.
Not Found
Insert into distance tree.
Insert <L_i, T_i> into set S.
If |S| > s_{max}:
Dequeue entries <L_j, T_{max}>
Lower global threshold T=T_{max}
Remove L_j from distance tree
Remove L_j from hash table
Priority queue for fast dequeue
Sample set S
Tree for fast distance computation
Update histogram with distance D_i
Rescale D_i before updating:
D_i=D_i / R, where R=T/P.
Reuse distance histogram
Key:
Boxes show new steps in SHARDS compared to exact MRC construction
Specific to SHARDS (both variants)
Specific to Fixed Size SHARDS
L_i = reference location
P = modulus
T = adaptive global threshold
T_i = threshold for L_i= hash(L_i) mod P
T_{max}= largest T among samples in S

Figure 2: **Algorithm Overview.** New steps in SHARDS compared to a standard exact MRC construction algorithm.

T dynamically. When the threshold is lowered from T to T', a *subset-inclusion property* is maintained automatically. Each location sampled *after* lowering the rate would also have been sampled *prior* to lowering the rate; since $T' < T$, the samples selected with T' are a proper subset of those selected with T.

2.2 Fixed-Rate MRC Construction

Conventional reuse-distance algorithms construct an exact MRC from a complete reference trace [34, 39]. Conveniently, as shown in Figure 2, existing MRC construction implementations can be run, essentially unmodified, by providing them a sampled reference trace as input. The only modification is that each reuse distance must be scaled appropriately by $1/R$, since each sampled location statistically represents a larger number of locations.

Standard MRC construction algorithms are computationally expensive. Consider a reference stream containing N total references to M unique locations. While an optimized implementation using efficient data structures requires only $O(N \log M)$ time, it still consumes $O(M)$ space for the hash table and balanced tree used to compute reuse distances. SHARDS can be used to construct an approximate MRC in dramatically less time and space. With a fixed sampling rate R, the expected number of unique sampled locations becomes $R \cdot M$. Assuming the sampled locations are fairly representative, the total number of sampled references is reduced to approximately $R \cdot N$. As we will see in Section 4, for most workloads, $R = 0.001$ yields very accurate MRCs, using memory and processing resources that are orders of magnitude smaller than conventional approaches.

2.3 Fixed-Size MRC Construction

Fixed-*rate* MRC construction achieves a radical reduction in computational resource requirements. Nevertheless, even with a low, constant sampling rate, space requirements may still grow without bound, along with the total number of unique locations that must be tracked. For memory-constrained environments, such as production cache controller firmware where MRCs could inform cache allocation decisions, it is desirable to place an upper bound on memory size.

An additional issue is the choice of an appropriate sampling rate, R, since the accuracy of MRC approximation using spatial sampling also depends on N and M. When these values are small, it is preferable to use a relatively large value for R (such as 0.1) to improve accuracy. When these values are large, it is preferable to use a relatively small value of R (such as 0.001), to avoid wasting or exhausting available resources. Weighing these tradeoffs is difficult, especially with incomplete information.

This suggests that accuracy may depend more on an adequate sample *size* than a particular sampling *rate*. This motivates an extended version of SHARDS that constructs an MRC in $O(1)$ space and $O(N)$ time, regardless of the size or other properties of its input trace.

2.3.1 Sampling Rate Adaptation

An appropriate sampling rate is determined automatically, and need not be specified. The basic idea is to lower the sampling rate adaptively, in order to maintain a fixed bound on the total number of sampled locations that are tracked at any given point in time. The sampling rate is initialized to a high value, and is lowered gradually as more unique locations are encountered. This approach leverages the subset-inclusion property maintained by SHARDS as the rate is reduced.

Initially, the sampling rate is set to a high value, such as $R_0 = 1.0$, the maximum possible value. This is implemented by using a sampling condition of the form $hash(L) \bmod P < T$, and setting the initial threshold $T = P$, so that every location L will be selected. In practice, $R_0 = 0.1$ is sufficiently high for nearly any workload.

The goal of operating in constant space implies that we cannot continue to track all sampled references. As shown in Figure 2, a new auxiliary data structure is introduced to maintain a fixed-size set S with cardinality $|S|$. Each element of S is a tuple $\langle L_i, T_i \rangle$, consisting of an actively-sampled location L_i, and its associated threshold value, $T_i = hash(L_i) \bmod P$. Let s_{max} denote the maximum desired size $|S|$ of set S; *i.e.*, s_{max} is a constant representing an upper bound on the number of actively-sampled locations. S can be implemented efficiently as a priority queue, ordered by the tuple's threshold value.

When the first reference to a location L that satisfies the current sampling condition is processed, it is a *cold miss*, since it has never been resident in the cache. In this case, L is not already in S, so it must be added to the set. If, after adding L, the bound on the set of active locations would be exceeded, such that $|S| > s_{max}$, then the size of S must be reduced. The element $\langle L_j, T_{max} \rangle$ with the largest threshold value T_{max} is removed from the set, using a priority-queue dequeue operation. The threshold T used in the current sampling condition is reduced to T_{max}, effectively reducing the sampling rate from $R_{old} = T/P$ to a new, strictly lower rate $R_{new} = T_{max}/P$, narrowing the criteria used for future sample selection.

The corresponding location L_j is also removed from all other data structures, such as the hash table and tree used in standard implementations. If any additional elements of S have the same threshold T_{max}, then they are also removed from S in the same manner.

2.3.2 Histogram Count Rescaling

As with fixed-rate sampling, reuse distances must be scaled by $1/R$ to reflect the sampling rate. An additional consideration for the fixed-size case is that R is adjusted dynamically. As the rate is reduced, the counts associated with earlier updates to the reuse-distance histogram need to be adjusted. Ideally, the effects of all updates associated with an evicted sample should be rescaled exactly. Since this would incur significant space and processing costs, we opt for a simple approximation.

When the threshold is reduced, the count associated with each histogram bucket is scaled by the ratio of the new and old sampling rates, R_{new}/R_{old}, which is equivalent to the ratio of the new and old thresholds, T_{new}/T_{old}. Rescaling makes the simplifying assumption that previous references to an evicted sample contributed equally to all existing buckets. While this is unlikely to be true for any individual sample, it is nonetheless a reasonable statistical approximation when viewed over many sample evictions and rescaling operations. Rescaling ensures

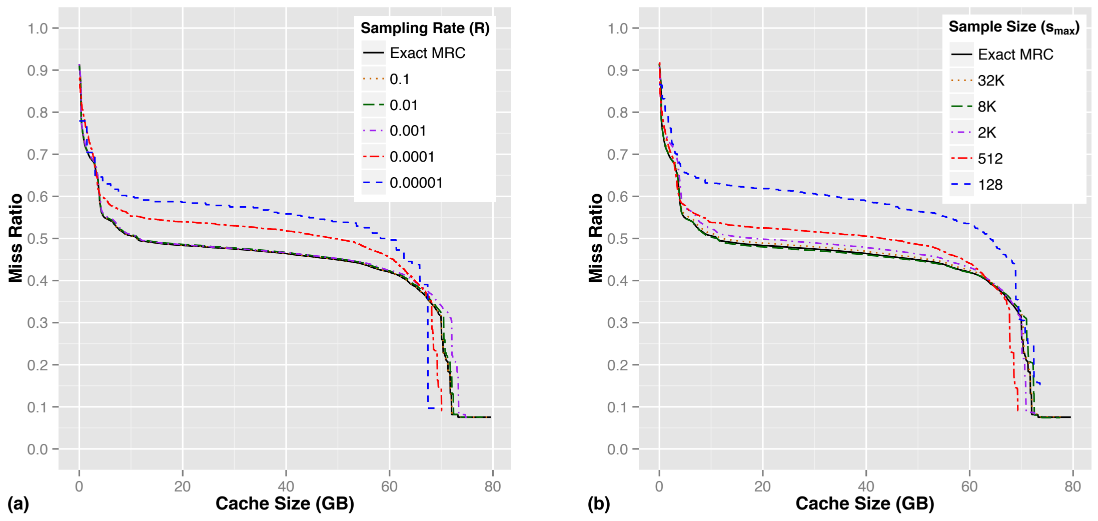


Figure 3: **Example SHARDS MRCs.** MRCs constructed for a block I/O trace containing 69.5M references to 5.2M unique blocks, using (a) fixed-rate SHARDS, varying R from 0.00001 to 0.1, and (b) fixed-size SHARDS, varying s_{max} from 128 to 32K.

that subsequent references to the remaining samples in S have the appropriate relative weight associated with their corresponding histogram bucket increments.

Conceptually, rescaling occurs immediately each time the current sampling threshold T is reduced. In practice, to avoid the expense of rescaling all histogram counts on every threshold change, it is instead performed incrementally. This is accomplished efficiently by storing T_{bucket} with each histogram bucket, representing the sampling threshold in effect when the bucket was last updated. When incrementing a bucket count, if $T_{bucket} \neq T$, then the existing count is first rescaled by T/T_{bucket}, the count is incremented, and T_{bucket} is set to T. During the final step in MRC construction, when histogram buckets are summed to generate miss ratios, any buckets for which $T_{bucket} \neq T$ need to be similarly rescaled.

3 Design and Implementation

We have developed several different implementations of SHARDS. Although designed for flexible experimentation, efficiency — especially space efficiency — was always a key goal. This section describes important aspects of both our fixed-rate and fixed-size MRC construction implementations, and discusses considerations for modeling various cache policies.

3.1 Fixed-Rate Implementation

To facilitate comparison with a known baseline, we start with the sequential version of the open-source C implementation of PARDA [39, 38]. PARDA takes a trace file as input, and performs offline reuse distance analysis, yielding an MRC. The implementation leverages two key data structures: a hash table that maps a location to the timestamp of its most recent reference, and a splay tree [48, 47] that is used to compute the number of distinct locations referenced since this timestamp.

Only a few simple modifications to the PARDA code were required to implement fixed-rate SHARDS, involving less than 50 lines of code. First, each referenced location read from the trace file is hashed, and processed only if it meets the specified sampling condition $hash(L) \bmod P < T$. For efficiency, the modulus P is set to a power of two[2] and "mod P" is replaced with the less expensive bitwise mask operation "& $(P-1)$". For a given sampling rate R, the threshold T is set to $round(R \cdot P)$. For the hash function, we used the public-domain C implementation of MurmurHash3 [3]. We also experimented with other hash functions, including a fast pseudo-random number generator [13], and found that they yielded nearly identical results.

Next, computed reuse distances are adjusted to reflect the sampling rate. Each raw distance D is simply divided by R to yield the appropriately scaled distance D/R. Since $R = T/P$, the scaled distance $(D \cdot P)/T$ is computed efficiently using an integer shift and division.

Figure 3(a) presents an example application of fixed-rate SHARDS, using a real-world storage block I/O trace[3]. The exact MRC is constructed using the unsampled, full-trace PARDA baseline. Five approximate MRCs are plotted for different fixed sampling rates, varying R between 0.00001 and 0.1, using powers of ten. The approximate curves for $R \geq 0.001$ are nearly indistinguishable from the exact MRC.

[2]We use $P = 2^{24}$, providing sufficient resolution to represent very low sampling rates, while still avoiding integer overflow when using 64-bit arithmetic for scaling operations.

[3]Customer VM disk trace *t*04, which also appears later in Figure 5.

Data structure element	$s_{max} <$ **64K**	$s_{max} <$ **4G**
hash table chain pointer	2	4
hash table entry	12	16
reference splay tree node	14	20
sample splay tree node	12	20
total per-sample size	40	60

Table 1: **Fixed-size SHARDS Data Structure Sizes.** Size (in bytes) used to represent elements of key data structures, for both 16-bit and 32-bit values of s_{max}.

3.2 Fixed-Size Implementation

With a constant memory footprint, fixed-size SHARDS is suitable for online use in memory-constrained systems, such as device drivers in embedded systems. To explore such applications, we developed a new implementation, written in C, optimized for for space efficiency.

Since all data structure sizes are known up-front, memory is allocated only during initialization. In contrast, other implementations perform a large number of dynamic allocations for individual tree nodes and hash table entries. A single, contiguous allocation is faster, and enables further space optimizations. For example, if the maximum number of samples s_{max} is bounded by 64K, "pointers" can be represented compactly as 16-bit indices instead of ordinary 64-bit addresses.

Like PARDA, our implementation leverages Sleator's public-domain splay tree code [47]. In addition to using a splay tree for computing reuse distances, we employ a second splay tree to maintain a priority queue representing the sample set S, ordered by hash threshold value. A conventional chained hash table maps locations to splay tree nodes. As an additional space optimization, references between data structures are encoded using small indices instead of general-purpose pointers.

The combined effect of these space-saving optimizations is summarized in Table 1, which reports the per-sample sizes for key data structures. Additional memory is needed for the output histogram; each bucket consumes 12 bytes to store a count and the update threshold T_{bucket} used for rescaling. For example, with s_{max} = 8K, the aggregate overhead for samples is only 320 KB. Using 10K histogram buckets, providing high resolution for evaluating cache allocation sizes, consumes another 120 KB. Even when code size, stack space, and all other memory usage is considered, the entire measured runtime footprint remains smaller than 1 MB, making this implementation practical even for extremely memory-constrained execution environments.

Figure 3(b) presents an example application of fixed-size SHARDS, using the same trace as Figure 3(a). Five approximate MRCs are plotted for different fixed sample sizes, varying s_{max} between 128 and 32K, using factors of four. The approximate curves for $s_{max} \geq$ 2K are nearly indistinguishable from the exact MRC.

3.3 Modeling Cache Policy

PARDA uses a simple binary trace format: a sequence of 64-bit references, with no additional metadata. Storage I/O traces typically contain richer information for each reference, including a timestamp, access type (read or write), and a location represented as an offset and length.

For the experiments in this paper, we converted I/O block traces to the simpler PARDA format, assumed a fixed cache block size, and ignored the distinction between reads and writes. This effectively models a simple LRU policy with fixed access granularity, where the first access to a block is counted as a miss.

We have also developed other SHARDS implementations to simulate diverse caching policies. For example, on a write miss to a partial cache block, a write-through cache may first read the entire enclosing cache-block-sized region from storage. The extra read overhead caused by partial writes can be modeled by maintaining separate histograms for ordinary reads and reads induced by partial writes. Other write-through caches manage partial writes at sub-block granularity, modeled using known techniques [57]. In all cases, we found hash-based spatial sampling to be extremely effective.

4 Experimental Evaluation

We conducted a series of experiments with over a hundred real-world I/O traces collected from our commercial caching analytics service for virtualized environments. We first describe our data collection system and characterize the trace files used in this paper. Next, we evaluate the accuracy of approximate MRCs. Finally, we present results of performance experiments that demonstrate the space and time efficiency of our implementations.

4.1 Data Collection

Our SaaS caching analytics service is designed to collect block I/O traces for VMware virtual disks in customer data centers running the VMware ESXi hypervisor [60]. A user-mode application, deployed on each ESXi host, coordinates with the standard VMware *vscsiStats* utility [1] to collect complete block I/O traces for VM virtual disks. A web-based interface allows particular virtual disks to be selected for tracing remotely.

Compressed traces are streamed to a cloud-based backend to perform various storage analyses, including offline MRC construction using SHARDS. If the trace is not needed for additional storage analysis, SHARDS sampling could be performed locally, obviating the need to stream full traces. Ideally, SHARDS should be integrated directly with the kernel-mode hypervisor component of *vscsiStats* for maximum efficiency, enabling continuous, online reuse-distance analysis.

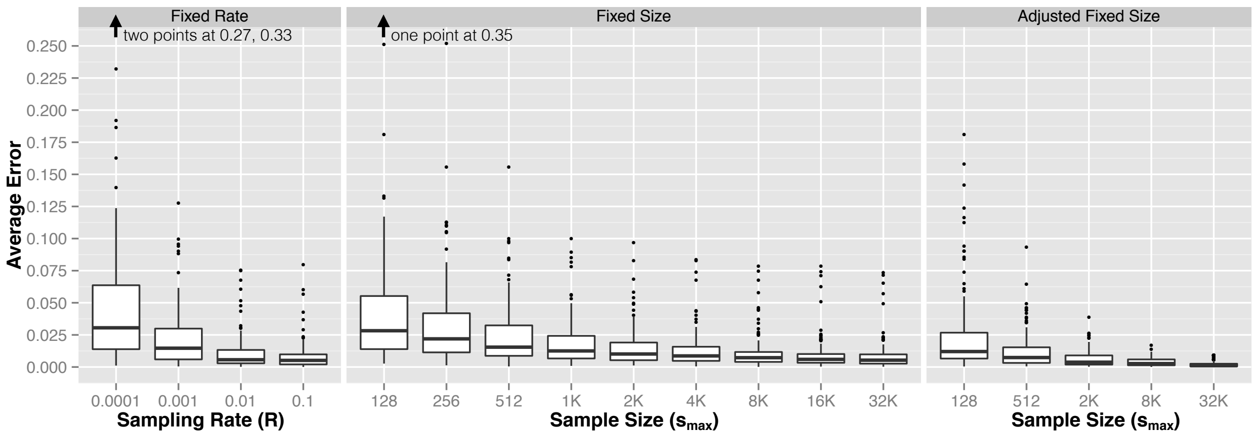


Figure 4: **Error Analysis.** Average absolute error calculated over all 124 traces for different SHARDS sampling parameters. The fixed-rate and fixed-size results are explained in Section 4.3.2, and the adjusted fixed-size results are discussed in Section 4.3.3.

4.2 Trace Files

We use 106 week-long vscsiStats traces, collected by our caching analytics service from virtual disks in production customer environments. These traces represent VMware virtual disks with sizes ranging from 8 GB to 34 TB, with a median of 90 GB. The associated VMs are a mix of Windows and Linux, with up to 64 GB RAM (6 GB median) and up to 32 virtual CPUs (2 vCPUs median).

In addition, we included several publicly-available block I/O traces from the SNIA IOTTA repository [51]. We used a dozen week-long enterprise server traces collected by Microsoft Research Cambridge [37], as well as six day-long server traces collected by FIU [31]. In total, this gives us a diverse set of 124 real-world block I/O traces to evaluate the accuracy and performance of SHARDS compared to exact methods.

4.3 Accuracy

We analyze the accuracy of MRCs constructed using SHARDS by comparing them to corresponding exact MRCs without sampling. Differences between the approximate and exact curves are measured over a wide range of sampling parameters. Numerous MRC plots are displayed as visual examples of SHARDS' accuracy.

4.3.1 Parameters

Our system supports many configuration parameters. We specify a 16 KB cache block size, so that a cache miss reads from primary storage in aligned, fixed-size 16 KB units; typical storage caches in commercial virtualized systems employ values between 4 KB and 64 KB. As discussed in Section 3.3, reads and writes are treated identically, effectively modeling a simple LRU cache policy. By default, we specify a histogram bucket size of 4K cache blocks, so that each bucket represents 64 MB.

Fixed-rate sampling is characterized by a single parameter, the sampling rate R, which we vary between 0.0001 and 0.1 using powers of ten. Fixed-size sampling has two parameters: the sample set size, s_{max}, and the initial sampling rate, R_0. We vary s_{max} using powers of two between 128 and 32K, and use $R_0 = 0.1$, since this rate is sufficiently high to work well with even small traces.

4.3.2 Error Metric

To analyze the accuracy of SHARDS, we consider the difference between each approximate MRC, constructed using hash-based spatial sampling, and its corresponding exact MRC, generated from a complete reference trace. An intuitive measure of this distance, also used to quantify error in related work [53, 43, 65], is the mean absolute difference or error (MAE) between the approximate and exact MRCs across several different cache sizes. This difference is between two values in the range [0, 1], so an absolute error of 0.01 represents 1% of that range.

The box plots[4] in Figure 4 show the MAE metric for a wide range of fixed-rate and fixed-size sampling parameters. For each trace, this distance is computed over all discrete cache sizes, at 64 MB granularity (corresponding to all non-zero histogram buckets).

Overall, the average error is extremely small, even for low sampling rates and small sample sizes. Fixed-rate sampling with $R = 0.001$ results in approximate MRCs with a median MAE of less than 0.02; most exhibit an MAE bounded by 0.05. The error for fixed-rate SHARDS typically has larger variance than fixed-size SHARDS, indicating that accuracy is better controlled via sample count than sampling rate.

For fixed-size SHARDS with s_{max} = 8K, the median MAE is 0.0072, with a worst-case of 0.078. Aside from a few outliers (13 traces), the error is bounded by 0.021.

[4]The top and the bottom of each box represent the first and third quartile values of the error. The thin whiskers represents the min and max error, excluding outliers. Outliers, represented by dots, are the values larger than $Q_3 + 1.5 \times IQR$, where $IQR = Q_3 - Q_1$.

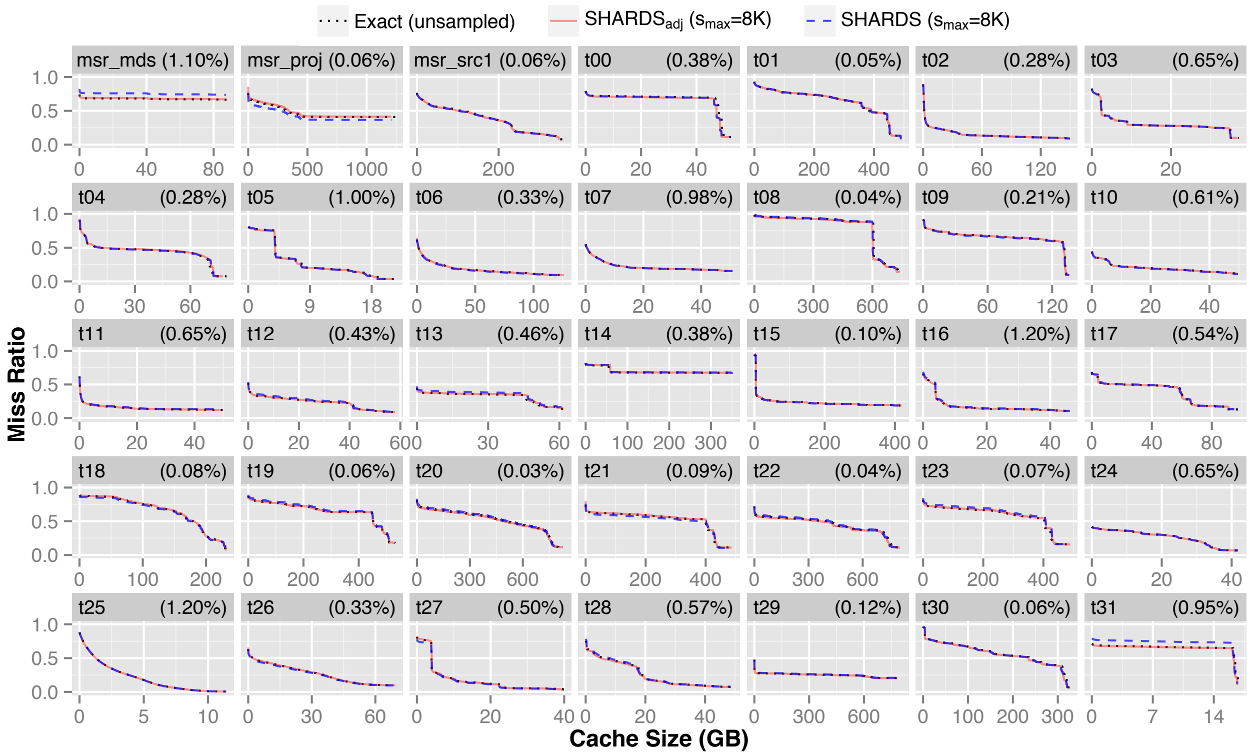


Figure 5: **Example MRCs: Exact vs. Fixed-Size SHARDS.** Exact and approximate MRCs for 35 representative traces. Approximate MRCs are constructed using fixed-size SHARDS and SHARDS$_{\text{adj}}$ with s_{max} = 8K. Trace names are shown for three public MSR traces [37]; others are anonymized as *t*00 through *t*31. The effective sampling rates appear in parentheses.

4.3.3 Using Reference Estimates to Reduce Error

In cases where SHARDS exhibits non-trivial error, we find that a coarse "vertical shift" accounts for most of the difference, while finer features are modeled accurately. This effect is seen in Figure 3. Recently, this observation led us to develop SHARDS$_{\text{adj}}$, a simple adjustment that improves accuracy significantly at virtually no cost.

Spatial sampling selects a static set of blocks. If the dynamic behavior of the sample set differs too much from that of the complete trace, the weights of the sums of buckets and the total count of accesses from the reuse histogram will be off, skewing the resulting MRC. For example, excluding too many or too few very hot blocks biases dynamic access counts.

Ideally, the average number of repetitions per block should be the same for both the sample set and the complete trace. This happens when the actual number of sampled references, N_s, matches the expected number, $E[N_s] = N \cdot R$. When this does not occur, we find that it is because the sample set contains the wrong proportion of frequently accessed blocks. Our correction simply adds the difference, $E[N_s] - N_s$, to the first histogram bucket before computing final miss ratios.

We now consider this adjustment to be best practice. Although there was insufficient time to update all of our earlier experiments, SHARDS$_{\text{adj}}$ results appear in Figures 4 and 5. Figure 4 reveals that the error with SHARDS$_{\text{adj}}$ is significantly lower. Across all 124 traces, this adjustment reduces the median fixed-size SHARDS error with s_{max} = 8K to 0.0027, and the worst-case to 0.017, factors of nearly 3× and 5×, respectively. Excluding the two outliers, MAE is bounded at 0.012. Even with just 128 samples, the median MAE is only 0.012.

4.3.4 Example MRCs

The quantitative error measurements reveal that, for nearly all traces, with fixed-size sampling at s_{max} = 8K, the miss ratios in the approximate MRCs deviate only slightly from the corresponding exact MRCs. Although space limitations prevent us from showing MRCs for *all* of the traces described in Section 4.2, we present a large number of small plots for this practical configuration.

Figure 5 plots 35 approximate MRCs, together with the matching exact curves; in most cases, the curves are nearly indistinguishable. In all cases, the location of prominent features, such as steep descents, appear faithful. Each plot is annotated with the effective dynamic sampling rate, indicating the fraction of IOs processed, including evicted samples. This rate reflects the amount of processing required to construct the MRC.

SHARDS$_{\text{adj}}$ effectively corrects all cases with visible error. For trace *t*31, the worst case over all 124 traces for SHARDS, error is reduced from 0.078 to 0.008.

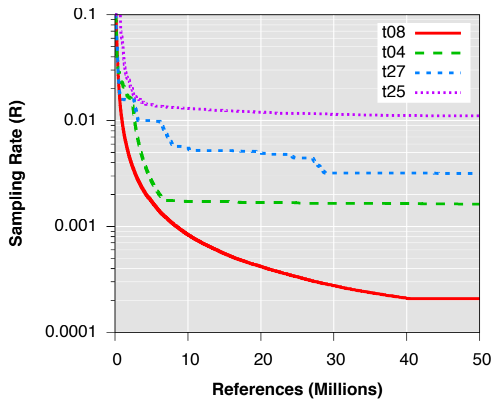


Figure 6: **Dynamic Rate Adaptation.** Sampling rate R (on log scale) for four traces over time. Each starts at $R_0 = 0.1$, and is lowered dynamically as more unique references are sampled.

4.3.5 Sampling Rate Adaptation

Choosing a sampling rate that achieves high accuracy with good efficiency is challenging. The automatic rate adaptation of fixed-size SHARDS is advantageous because it eliminates the need to specify R. Figure 6 plots R as a function of reference count for four diverse traces: *t08*, *t04*, *t27*, and *t25* from Figure 5. For each, the sampling rate starts at a high initial value of $R_0 = 0.1$, and is lowered progressively as more unique locations are encountered. The figure shows that SHARDS adapts automatically for each of the traces, which contain significantly different numbers of unique references. After 50 million references, the values of R for these traces are 0.0002, 0.0016, 0.0032, and 0.0111. The total number of samples processed, including evictions from the fixed-size sample set S, is given by the area under each curve.

4.3.6 Discussion

Quantitative experiments confirm that, for nearly all workloads, SHARDS yields accurate MRCs, in radically less time and space than conventional exact algorithms. While the accuracy achieved with high sampling rates may not be surprising, success with very low rates, such as $R = 0.001$, was unexpected. Even more extraordinary is the ability to construct accurate MRCs for a broad range of workloads, using only a small constant number of samples, such as s_{max} = 8K, or even s_{max} = 256. The constant-space and rate-adaptation properties of fixed-size SHARDS make it our preferred approach.

Our intuition is that most workloads are composed of a fairly small number of basic underlying processes, each of which operates somewhat uniformly over relatively large amounts of data. As a result, a small number of representative samples is sufficient to model the main underlying processes. Additional samples are needed to properly capture the relative weights of these processes. Interestingly, the number of samples required to obtain accurate results for a given workload may be indicative of its underlying dimensionality or intrinsic complexity.

Many statistical methods exhibit sampling error inversely proportional to $\sqrt{n}$, where n is the sample size. Our data is consistent; regressing the average absolute error for each s_{max} value shown in Figure 4 against $1/\sqrt{s_{max}}$ resulted in a high correlation coefficient of $r^2 = 0.97$. This explains the observed diminishing accuracy improvements with increasing s_{max}.

4.4 Performance

We conducted performance experiments in a VMware virtual machine, using a 64-bit Ubuntu 12.04 guest running Linux kernel version 3.2.0. The VM was configured with 64 GB RAM, and 8 virtual cores, and executed on an under-committed physical host running VMware ESXi 5.5, configured with 128 GB RAM and 32 AMD Opteron x86-64 cores running at 2 GHz.

To quantify the performance advantages of SHARDS over exact MRC construction, we use a modern high-performance reuse-distance algorithm from the open-source PARDA implementation [39, 38] as our baseline. Although the main innovation of PARDA is a parallel reuse distance algorithm, we use the same sequential "classical tree-based stack distance" baseline as in their paper. The PARDA parallelization technique would also result in further performance gains for SHARDS.

4.4.1 Space

To enable a fair comparison of memory consumption with SHARDS, we implemented minor extensions to PARDA, adding command-line options to specify the number of output histogram buckets and the histogram bucket width.[5] We also added code to both PARDA and SHARDS to obtain accurate runtime memory usage[6].

All experiments were run over the full set of traces described in Section 4.2. Each run was configured with 10 thousand histogram buckets, each 64 MB wide (4K cache blocks of size 16 KB), resulting in an MRC for cache allocations up to 640 GB.

Sequential PARDA serves as a baseline, representing an efficient, exact MRC construction algorithm without sampling. Fixed-rate SHARDS, implemented via the simple code modifications described in Section 3.1, is configured with $R = 0.01$ and $R = 0.001$. Finally, the new space-efficient fixed-size SHARDS implementation, presented in Section 3.2, is run with s_{max} = 8K and $R_0 = 0.1$.

Figure 7 shows the memory usage for each algorithm over the full set of traces, ordered by baseline memory consumption. The drastic reductions with SHARDS required the use of a log scale. As expected, for traces

[5]By default, PARDA is configured with hard-coded values – 1M buckets, each a single cache block wide.

[6]We obtain the peak resident set size directly from the Linux procfs node `/proc/<pid>/status` immediately before terminating; the `VmHWM` line reports the "high water mark" [32].

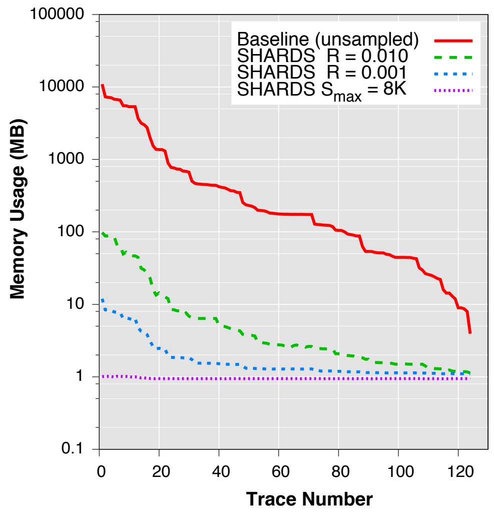


Figure 7: **Memory Usage.** Measured memory consumption (in MB, log scale) for unsampled baseline, fixed-rate SHARDS (R = 0.01, 0.001), and fixed-size SHARDS (s_{max} = 8K).

with large numbers of unique references, the memory required for fixed-rate SHARDS is approximately R times as big as the baseline. With much smaller traces, fixed overheads dominate. For fixed-size SHARDS, the runtime footprint remained approximately 1 MB for all runs, ranging from 964 KB to 1,044 KB, with an average of 974 KB, yielding a savings of up to $10,800\times$ for large traces and a median of $185\times$ across all traces.

4.4.2 Time

Figure 8 plots the CPU usage measured[7] for the same runs described above, ordered by baseline CPU consumption. The significant processing time reductions with SHARDS prompted the use of a log scale.

Fixed-rate SHARDS with $R = 0.01$ results in speedups over the baseline ranging from $29\times$ to $449\times$, with a median of $75\times$. For $R = 0.001$, the improvement ranges from $41\times$ to $1,029\times$, with a median of $128\times$. For short traces with relatively small numbers of unique references, fixed overheads dominate, limiting speedups to values lower than implied by R.

Fixed-size SHARDS with s_{max} = 8K and $R_0 = 0.1$ incurs more overhead than fixed-rate SHARDS with $R = 0.01$. This is due to the non-trivial work associated with evicted samples as the sampling rate adapts dynamically, as well as the cost of updating the sample set priority queue. Nonetheless, fixed-size SHARDS achieves significant speedups over the baseline, ranging from $6\times$ to $204\times$, with a median of $22\times$. In terms of throughput, for the top three traces ordered by CPU consumption in Figure 8, fixed-size SHARDS processes an average of 15.4 million references per second.

[7]For each run, CPU time was obtained by adding the user and system time components reported by `/usr/bin/time`.

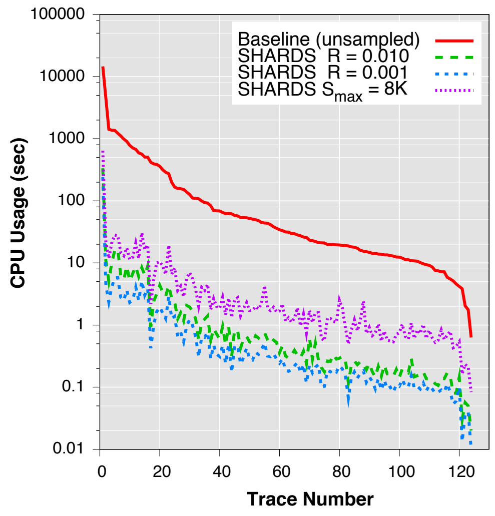


Figure 8: **CPU Usage.** Measured run time (in seconds, log scale) for unsampled baseline, fixed-rate SHARDS (R = 0.01, 0.001), and fixed-size SHARDS (s_{max} = 8K).

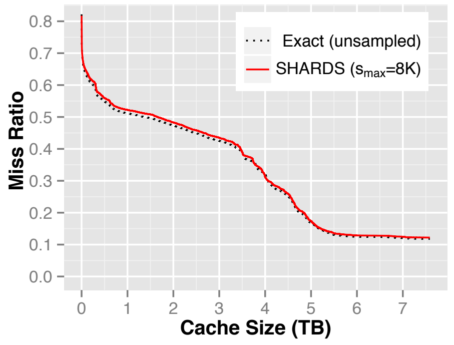


Figure 9: **Mixed Workloads.** Exact and approximate MRCs for merged trace interleaving 4.3G IOs to 509M unique blocks from 32 separate virtual disks. Fixed-size SHARDS with s_{max} = 8K exhibits an average absolute error of only 0.008.

4.5 MRCs for Mixed Workloads

Our VM-based traces represent single-machine workloads, while the IOs received by storage arrays are typically an undistinguished, blended mix of numerous independent workloads. Figure 9 demonstrates the accuracy of fixed-size SHARDS using a relative-time-interleaved reference stream combining all 32 virtual disk traces (*t00...t31*) shown in Figure 5. With s_{max} = 8K, SHARDS exhibits a small MAE of 0.008. The high accuracy and extremely low overhead provide additional confidence that continuous, online MRC construction and analysis is finally practical for production storage arrays.

4.6 Non-LRU Replacement Policies

SHARDS constructs MRCs for a cache using an LRU replacement policy. Significantly, the same underlying hash-based spatial sampling approach appears promising for simulating more sophisticated policies, such as LIRS [27], ARC [35], CAR [5], or Clock-Pro [26].

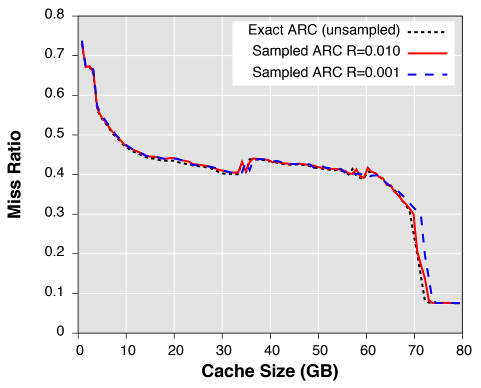


Figure 10: **Scaled-Down ARC Simulation.** Exact and approximate MRCs for VM disk trace *t*04. Each curve plots 100 separate ARC simulations at different cache sizes.

As with fixed-rate SHARDS, the input trace is filtered to select blocks that satisfy a hash-based sampling condition, corresponding to the sampling rate R. A series of separate simulations is run, each using a different cache size, which is also scaled down by R. Figure 10 presents results for the same trace as in Figure 3, leveraging an open-source ARC implementation [21]. For $R = 0.001$, the simulated cache is only 0.1% of the desired cache size, achieving huge reductions in space and time, while exhibiting excellent accuracy, with an MAE of 0.01.

5 Related Work

Before the seminal paper of Mattson, Gecsei, Slutz, and Traiger [34], studies of memory and storage systems required running separate experiments for each size of a given level of the hierarchy. Their key insight is that many replacement policies have an inclusion property: given a cache C of size z, $C(z) \subseteq C(z+1)$. Such policies, referred to as *stack algorithms*, include LRU, LFU, and MRU.[8] Mattson *et al.* also model set associativity, deletion, no-write-allocate policies, and the handling of the entire set of memory and storage hierarchies as a list of such stacks. Others extended the model for caches with write-back or subblocking policies [56, 57], variable-size pages [58], set associativity [24, 30, 52], and modeling groups of these behaviors in a single analysis [25, 59].

Because it generates models of behavior for all cache sizes in a single pass over a trace, Mattson's technique has been applied widely. Application areas include the modeling of caches [24, 30, 52]; of multicore caches including the effects of invalidation [45, 44]; guidance of mechanisms to apportion shared caches amongst processes [41, 53, 18]; the scheduling of memory within an operating system [49, 72, 4]; the sizing and management of unified buffer caches [28]; secondary exclusive (victim I/O) caches [33], and memory caches [43]; the sizing of garbage-collected heaps [68, 67]; the impact of memory systems and caches on Java performance [29]; the transparent borrowing of memory for low-priority computation [14]; the balancing of memory across sets of virtual machines [70, 69]; and the analysis of program behavior and compilation for data layout [2, 11, 17].

[8]Policies other than LRU require one more step: after a block is moved to the top, the remaining blocks are reordered in a single pass.

5.1 Optimizations

Mattson's algorithm takes $O(NM)$ time and $O(M)$ space for a trace of length N containing M unique blocks. Given its broad applicability, much effort has been spent improving its performance in both space and time.

5.1.1 Management of LRU Stacks

Early improvements added hash tables either to detect cold accesses or to map references to their previous entries [6, 40, 56]. Bennett and Kruskal [6] used a balanced tree over the trace tracking which references were the most recent instances and keeping counts in the subtrees, allowing each distance to be computed in $O(\log N)$ steps. Olken [40] improved on this by tracking only the most recent references in the tree, further reducing the distance computation to $O(\log M)$. Use of a balanced tree is now common for computing distances. By managing the stack of references with a doubly-linked or chunked list and mapping references to nodes in the stack, the algorithm takes $O(N \log M)$ time and $O(M)$ space.

Another key advance, by Niu *et al.*, is processing a trace in parallel [39]. Their technique, PARDA, achieves impressive speedups by splitting a trace into P partitions, processing each independently, and patching up missing information between partitions.

5.1.2 Compression and Grouping of References

Much effort has been spent reducing trace sizes [50, 36]. Smith [50] compressed virtual-memory traces by ignoring references whose reuse distance is less than some K, and periodically scanning access-bits for pages.

Another optimization is to coarsen the distances that are tracked [52, 72, 68, 67, 4, 70, 43]. Kim *et al.* [30] track groups of references in the stack where the sizes of the groups are powers of two. By tracking the boundary of each group of references, updates to the LRU stack simply adjust the distances for the pages pushed across these boundaries. This can reduce costs of tracking accesses to $O(G)$ where G is the number of groups tracked. Recently, Saemundsson *et al.* [43] grouped references into variable-sized buckets. Their ROUNDER aging algorithm with 128 buckets yields MAEs up to 0.04 with a median MAE of 0.006 for partial MRCs [42], but the space complexity remains $O(M)$. Zhao *et al.* [70] report an error rate of 10% for their level of coarseness, 32 pages. None of the others report error rates.

Ding and Zhong [17] apply clustering in the context of the splay tree they use to track reuse-distances for program analysis. By dynamically compressing the tree, they bound the overall size and cost of their analysis, achieving a time bound of $O(N \log \log M)$ and a space bound of $O(\log M)$. The relative error is bounded by how the nodes in the tree are merged and compressed and, so, factors in as the base of the log. For an error of 50%, the base is 2; for smaller ones, the base quickly approaches 1. For their purpose, they can tolerate large error bounds.

5.1.3 Temporal Sampling

Temporal sampling — complementary to SHARDS — reduces reference-tracking costs by only doing so some of the time. Berg *et al.* [7, 8] sample every *N*th reference (1 in every 10K) to derive MRCs for caches. Bryan and Conte's cluster sampling [10], RapidMRC [53], and work on low-cost tracking for VMs [69], by contrast, divide the execution into periods in which references are either sampled or are not. They also tackle how to detect phase changes that require regeneration of the reuse-distances. RapidMRC reports a mean average error rate of 1.02 misses per thousand instructions (MPKI) with a maximum of 6.57 MPKI observed. Zhao *et al.* [69] report mean relative errors of 3.9% to 12.6%. These errors are significantly larger than what SHARDS achieves.

Use of sampling periods allows for accurate measurements of reuse distances within a sample period. However, Zhong and Chang [71] and Schuff *et al.* [45, 44] observe that naively sampling every *N*th reference as Berg *et al.* do or using simple sampling phases causes a bias against the tracking of longer reuse distances. Both efforts address this bias by sampling references during a sampling period and then following their next accesses across subsequent sampling and non-sampling phases.

5.1.4 Spatial and Content-Based Sampling

A challenge when sampling references is that reuse-distance is a recurrent behavior. One solution is to extract a sample from the trace based on an identifying characteristic of its references. Spatial sampling uses addresses to select a sample set. Content-based sampling does so by using data contents. Both techniques can capture all events for a set of references, even those that occur rarely.

Many analyses for set-associative caches have used set-sampling [23, 41, 46]. For example, UMON-DSS [41] reduces the cost of collecting reuse-distances by sampling the behavior of a subset of the sets in a processor cache. Kessler *et al.* [23] compare temporal sampling, set-sampling and constant-bit sampling of references and find that the last technique is most useful when studying set-associative caches of different dimensions.

Many techniques targeting hardware implementations use grouping or spatial sampling to constrain their use of space [72, 41, 4, 59, 46]. However, these tend to focus on narrow problems such as limited set associativity [41] or limited cache size ranges [4] for each MRC. Like these approaches, SHARDS reduces and bounds space use, but unlike them, it models the full range of cache sizes. In addition, these techniques do not report error rates.

Inspired by processor hardware for cache sampling, Waldspurger *et al.* propose constructing an MRC by sampling a fixed set of pages from the guest-physical memory of a VM [62]. Unfortunately, practical sampling requires using small (4 KB) pages, increasing the overhead of memory virtualization [9]. Choosing sampled locations up-front is also inefficient, especially for workloads with large, sparse address spaces. In contrast, SHARDS does not require any information about the address space.

Xie *et al.* address a different problem: estimation of duplication among blocks in a storage system [66]. Their system hashes the contents of blocks producing fingerprints. These are partitioned into sets with one set chosen as the sample. Their model has error proportional to the sample-set size. This property is used to dynamically repartition the sample so that the sample size is bounded. Like Xie *et al.*, the SHARDS sampling rate can be adjusted to ensure an upper bound on the space used. But, how the sample set is chosen, how the sampling rate is adjusted, and how the sampling ratio is used to adjust the summary information are different. Most importantly, where their work looks at individual blocks' hash values and how these collide, our technique accurately captures the relationship *between* pairs of *accesses* to the blocks.

5.2 Analytical Models

Many analytical models have been proposed to approximate MRCs with reduced effort. By constraining the block replacement policy, Tay and Zou [55] derive a universal equation that models cache behavior from a small set of sampled data points. He *et al.* propose modeling MRCs as fractals and claim error rates of 7-10% in many cases with low overhead [22]. Berg *et al.* [7, 8, 19, 18] use a closed-form equation of the miss rate. Through a sequence of sampling, deriving local miss rates and combining these separate curves, they model caches with random or LRU replacement. Others model cache behavior by tracking hardware performance counters [15, 63, 46].

Unlike the analytical approaches, SHARDS estimates the MRC directly from the sampled trace. We have shown that SHARDS can be implemented using constant space and with high accuracy. Where the error of SHARDS is small, the analytic techniques report errors of a few percent to 50% with some outliers at 100-200%. Berg *et al.* simply offer graphs for comparison.

5.3 Counter Stacks

Mattson *et al.* track distances as counts of unique references between reuses. Wires *et al.* extend this in three ways in their recent MRC approximation work, using a *counter stack* [65].

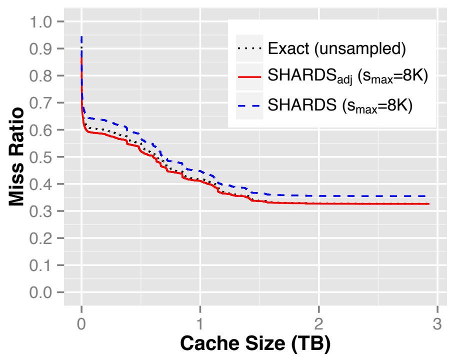


Figure 11: **Merged MSR Trace.** Exact, SHARDS and SHARDS$_{adj}$ MRCs for the merged "master" MSR trace used in the Counter Stacks evaluation [65], with s_{max} = 8K.

First, the counts of repetitions, themselves, can be computed by comparing changes in the number of unique references seen from different starting points in the stream. The sequence of locations observed by a newer counter is a proper suffix of the sequence recorded by an older one. So, if the newer counter increases but the older does not, then the older location must have repeated, and its reuse-distance is the older counter's value.

Second, the repetitions and reuse-distances can be approximated efficiently using a bounded set of counters. Instead of starting a new counter with every reference, one may *downsample* the set of counters, creating and tracking a new one periodically. The set can be further *pruned* since, over time, adjacent counters converge as they observe the same set of elements. Using probabilistic counters based on the HyperLogLog algorithm [20] together with downsampling and pruning, the counter stack algorithm only uses $O(\log M)$ space.

Third, columns of counts in the counter stack can be periodically written to a checkpoint together with timestamps for subsequent analysis. Checkpointed counter-stack sequences can be spliced, shifted temporally, and combined to model the behavior of combinations of workloads. Because the checkpoint only captures stacks of counts at each timestamp, such modeling assumes that different checkpoints access disjoint sets of blocks.

To provide a direct quantitative comparison with SHARDS, we generated the same merged "master" MSR trace used by Wires *et al.* [65], configured identically with only read requests and a 4 KB cache block size. Figure 11 shows MRCs constructed using fixed-size SHARDS, with 48K histogram buckets of size 64 MB, supporting cache sizes up to 3 TB. For s_{max} = 8K, the MAE is 0.006 with SHARDS$_{adj}$ (0.029 unadjusted). The MRC is computed using only 1.3 MB of memory in 137 seconds, processing 17.6M blocks/sec. Wires *et al.* report that Counter Stacks requires 80 MB of memory, and 1,034 seconds to process this trace at a rate of 2.3M blocks/sec. In this case, Counter Stacks is approximately 7$\times$ slower and needs 62$\times$ as much memory as SHARDS$_{adj}$, but is more accurate, with an MAE of only 0.0025 [64]. Using s_{max} = 32K, with 2 MB of memory in 142 seconds, yields a comparable MAE of 0.0026.

While Counter Stacks uses $O(\log M)$ space, fixed-size SHARDS computes MRCs in small *constant* space. As a result, separate SHARDS instances can efficiently compute multiple MRCs tracking different properties or time-scales for a given reference stream, something Wires *et al.* claim is not practical.

One advantage of Counter Stacks is that every reference affects the probabilistic counters and contributes to the resulting MRC. By contrast, SHARDS assumes that hashing generates a uniformly distributed set of values for a reference stream. While an adversarial trace could yield an inaccurate MRC, we have not encountered one.

Unlike Counter Stacks, SHARDS maintains the identity of each block in its sample set. This enables tracking additional information, including access frequency, making it possible to directly implement other policies such as LFU, LIRS [27], ARC [35], CAR [5], or Clock-Pro [26], as discussed in Section 4.6.

6 Conclusions

We have introduced SHARDS, a new hash-based spatial sampling technique for reuse-distance analysis that computes approximate miss ratio curves accurately using only modest computational resources. The approach is so lightweight — operating in constant space, and requiring several orders of magnitude less processing than conventional algorithms — that online MRC construction becomes practical. Furthermore, SHARDS enables offline analysis for long traces that, due to memory constraints, could not be studied using exact techniques.

Our experimental evaluation of SHARDS demonstrates its accuracy, robustness, and performance advantages, over a large collection of I/O traces from real-world production storage systems. Quantitative results show that, for most workloads, an approximate sampled MRC that differs only slightly from an exact MRC can be constructed in 1 MB of memory. Performance analysis highlights dramatic reductions in resource consumption, up to $10,800\times$ in memory and up to $204\times$ in CPU.

Encouraged by progress generalizing hash-based spatial sampling to model sophisticated replacement policies, such as ARC, we are exploring similar techniques for other complex systems. We are also examining the rich temporal dynamics of MRCs at different time scales.

Acknowledgments Thanks to the anonymous reviewers, our shepherd Arif Merchant, Jim Kleckner, Xiaojun Liu, Guang Yang, John Blumenthal, and Jeff Hausman for their valuable feedback and support.

References

[1] AHMAD, I. Easy and efficient disk I/O workload characterization in VMware ESX Server. In *Proceedings of the 2007 IEEE 10th International Symposium on Workload Characterization* (Washington, DC, USA, 2007), IISWC '07, IEEE Computer Society, pp. 149–158.

[2] ALMÁSI, G., CAŞCAVAL, C., AND PADUA, D. A. Calculating stack distances efficiently. *SIGPLAN Not. 38*, 2 supplement (June 2002), 37–43.

[3] APPLEBY, A. SMHasher and MurmurHash. `https://code.google.com/p/smhasher/`.

[4] AZIMI, R., SOARES, L., STUMM, M., WALSH, T., AND BROWN, A. D. Path: Page access tracking to improve memory management. In *Proceedings of the 6th International Symposium on Memory Management* (New York, NY, USA, 2007), ISMM '07, ACM, pp. 31–42.

[5] BANSAL, S., AND MODHA, D. S. CAR: Clock with adaptive replacement. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2004), FAST '04, USENIX Association, pp. 187–200.

[6] BENNETT, B. T., AND KRUSKAL, V. J. LRU stack processing. *IBM Journal of Research and Development 19* (1975), 353–357.

[7] BERG, E., AND HAGERSTEN, E. StatCache: A Probabilistic Approach to Efficient and Accurate Data Locality Analysis. In *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2004)* (Austin, Texas, USA, Mar. 2004).

[8] BERG, E., AND HAGERSTEN, E. Fast Data-Locality Profiling of Native Execution. In *Proceedings of ACM SIGMETRICS 2005* (Banff, Canada, June 2005).

[9] BHARGAVA, R., SEREBRIN, B., SPADINI, F., AND MANNE, S. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2008), ASPLOS XIII, ACM, pp. 26–35.

[10] BRYAN, P. D., AND CONTE, T. M. Combining cluster sampling with single pass methods for efficient sampling regimen design. In *25th International Conference on Computer Design, ICCD 2007, 7-10 October 2007, Lake Tahoe, CA, USA, Proceedings* (2007), IEEE, pp. 472–479.

[11] CAŞCAVAL, C., AND PADUA, D. A. Estimating cache misses and locality using stack distances. In *Proceedings of the 17th Annual International Conference on Supercomputing* (New York, NY, USA, 2003), ICS '03, ACM, pp. 150–159.

[12] CARR, R. W., AND HENNESSY, J. L. WSCLOCK–a simple and effective algorithm for virtual memory management. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1981), SOSP '81, ACM, pp. 87–95.

[13] CARTA, D. F. Two fast implementations of the minimal standard random number generator. *CACM 33*, 1 (Jan. 1990), 87–88.

[14] CIPAR, J., CORNER, M. D., AND BERGER, E. D. Transparent contribution of memory. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference* (Berkeley, CA, USA, 2006), ATEC '06, USENIX Association, pp. 11–11.

[15] CONTE, T. M., HIRSCH, M. A., AND HWU, W.-M. W. Combining trace sampling with single pass methods for efficient cache simulation. *IEEE Trans. Comput. 47*, 6 (June 1998), 714–720.

[16] DENNING, P. J. The working set model for program behavior. *Commun. ACM 11*, 5 (May 1968), 323–333.

[17] DING, C., AND ZHONG, Y. Predicting whole-program locality through reuse distance analysis. *SIGPLAN Not. 38*, 5 (May 2003), 245–257.

[18] EKLOV, D., BLACK-SCHAFFER, D., AND HAGERSTEN, E. Fast modeling of shared caches in multicore systems. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers* (New York, NY, USA, 2011), HiPEAC '11, ACM, pp. 147–157.

[19] EKLOV, D., AND HAGERSTEN, E. StatStack: Efficient modeling of LRU caches. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on* (March 2010), pp. 55–65.

[20] FLAJOLET, P., FUSY, E., GANDOUET, O., AND MEUNIER, F. HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm. In *Proceedings of the 2007 International Conference on Analysis of Algorithms (AOFA '07)* (2007), pp. 127–146.

[21] GRYSKI, D. go-arc git repository. `https://github.com/dgryski/go-arc/`.

[22] HE, L., YU, Z., AND JIN, H. FractalMRC: Online cache miss rate curve prediction on commodity systems. In *IPDPS'12* (2012), pp. 1341–1351.

[23] HILL, K. M., KESSLER, R. E., HILL, M. D., AND WOOD, D. A. A comparison of trace-sampling techniques for multi-megabyte caches. *IEEE Transactions on Computers 43* (1994), 664–675.

[24] HILL, M. D., AND SMITH, A. J. Evaluating associativity in CPU caches. *IEEE Trans. Comput. 38*, 12 (Dec. 1989), 1612–1630.

[25] JANAPSATYA, A., IGNJATOVIĆ, A., AND PARAMESWARAN, S. Finding optimal L1 cache configuration for embedded systems. In *Proceedings of the 2006 Asia and South Pacific Design Automation Conference* (Piscataway, NJ, USA, 2006), ASP-DAC '06, IEEE Press, pp. 796–801.

[26] JIANG, S., CHEN, F., AND ZHANG, X. Clock-pro: An effective improvement of the clock replacement. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2005), ATEC '05, USENIX Association, pp. 35–35.

[27] JIANG, S., AND ZHANG, X. Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2002), SIGMETRICS '02, ACM, pp. 31–42.

[28] KIM, J. M., CHOI, J., KIM, J., NOH, S. H., MIN, S. L., CHO, Y., AND KIM, C. S. A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references. In *Proceedings of the 4th Symposium on Operating System Design and Implementation – Volume 4* (Berkeley, CA, USA, 2000), OSDI'00, USENIX Association, pp. 9–9.

[29] KIM, J.-S., AND HSU, Y. Memory system behavior of Java programs: Methodology and analysis. In *Proceedings of the 2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2000), SIGMETRICS '00, ACM, pp. 264–274.

[30] KIM, Y. H., HILL, M. D., AND WOOD, D. A. Implementing stack simulation for highly-associative memories. In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 1991), SIGMETRICS '91, ACM, pp. 212–213.

[31] KOLLER, R., AND RANGASWAMI, R. I/O deduplication: Utilizing content similarity to improve I/O performance. *Trans. Storage 6*, 3 (Sept. 2010), 13:1–13:26.

[32] LINUX PROGRAMMER'S MANUAL. proc(5) Linux manual page. `http://man7.org/linux/man-pages/man5/proc.5.html`.

[33] LU, P., AND SHEN, K. Multi-layer event trace analysis for parallel I/O performance tuning. *2013 42nd International Conference on Parallel Processing* (2007), 12.

[34] MATTSON, R. L., GECSEI, J., SLUTZ, D. R., AND TRAIGER, I. L. Evaluation techniques for storage hierarchies. *IBM Syst. J. 9*, 2 (June 1970), 78–117.

[35] MEGIDDO, N., AND MODHA, D. S. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2003), FAST '03, USENIX Association, pp. 115–130.

[36] MICHAUD, P. Online compression of cache-filtered address traces. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on* (April 2009), pp. 185–194.

[37] NARAYANAN, D., DONNELLY, A., AND ROWSTRON, A. Write off-loading: Practical power management for enterprise storage. *Trans. Storage 4*, 3 (Nov. 2008), 10:1–10:23.

[38] NIU, Q. PARDA git repository. `https://bitbucket.org/niuqingpeng/file_parda/`.

[39] NIU, Q., DINAN, J., LU, Q., AND SADAYAPPAN, P. PARDA: A fast parallel reuse distance analysis algorithm. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium* (Washington, DC, USA, 2012), IPDPS '12, IEEE Computer Society, pp. 1284–1294.

[40] OLKEN, F. Efficient methods for calculating the success function of fixed space replacement policies. *Perform. Eval. 3*, 2 (1983), 153–154.

[41] QURESHI, M. K., AND PATT, Y. N. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2006), MICRO 39, IEEE Computer Society, pp. 423–432.

[42] SAEMUNDSSON, T. Private communication, Jan 2015.

[43] SAEMUNDSSON, T., BJORNSSON, H., CHOCKLER, G., AND VIGFUSSON, Y. Dynamic performance profiling of cloud caches. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2014), SOCC '14, ACM, pp. 28:1–28:14.

[44] SCHUFF, D. L., KULKARNI, M., AND PAI, V. S. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques* (New York, NY, USA, 2010), PACT '10, ACM, pp. 53–64.

[45] SCHUFF, D. L., PARSONS, B. S., AND PAI, V. S. Multicore-aware reuse distance analysis. Tech. Rep. ECE-TR-388, Purdue University, Sep 2009.

[46] SEN, R., AND WOOD, D. A. Reuse-based online models for caches. In *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2013), SIGMETRICS '13, ACM, pp. 279–292.

[47] SLEATOR, D. An implementation of top-down splaying with sizes. `ftp://ftp.cs.cmu.edu/usr/ftp/usr/sleator/splaying`.

[48] SLEATOR, D. D., AND TARJAN, R. E. Self-adjusting binary search trees. *J. ACM 32*, 3 (July 1985), 652–686.

[49] SMARAGDAKIS, Y., KAPLAN, S. F., AND WILSON, P. R. The EELRU adaptive replacement algorithm. *Perform. Eval. 53*, 2 (2003), 93–123.

[50] SMITH, A. Two methods for the efficient analysis of memory address trace data. *Software Engineering, IEEE Transactions on SE-3*, 1 (Jan 1977), 94–101.

[51] SNIA. SNIA iotta repository block I/O traces. `http://iotta.snia.org/tracetypes/3`.

[52] SUGUMAR, R. A., AND ABRAHAM, S. G. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 1993), SIGMETRICS '93, ACM, pp. 24–35.

[53] TAM, D. K., AZIMI, R., SOARES, L. B., AND STUMM, M. RapidMRC: Approximating L2 miss rate curves on commodity systems for online optimizations. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2009), ASPLOS XIV, ACM, pp. 121–132.

[54] TANENBAUM, A. S. *Modern Operating Systems*, 3rd ed. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

[55] TAY, Y. C., AND ZOU, M. A page fault equation for modeling the effect of memory size. *Perform. Eval. 63*, 2 (Feb. 2006), 99–130.

[56] THOMPSON, J. G. *Efficient Analysis of Caching Systems*. PhD thesis, EECS Department, University of California, Berkeley, Sep 1987.

[57] THOMPSON, J. G., AND SMITH, A. J. Efficient (stack) algorithms for analysis of writeback and sector memories. *ACM Trans. Comput. Syst. 7*, 1 (Jan. 1989), 78–117.

[58] TRAIGER, I., AND SLUTZ, D. *One-pass Techniques for the Evaluation of Memory Hierarchies*. IBM research report. IBM Research Division, 1971.

[59] VIANA, P., GORDON-ROSS, A., BARROS, E., AND VAHID, F. A table-based method for single-pass cache optimization. In *Proceedings of the 18th ACM Great Lakes Symposium on VLSI* (New York, NY, USA, 2008), GLSVLSI '08, ACM, pp. 71–76.

[60] VMWARE, INC. VMware vSphere Hypervisor (ESXi). `http://www.vmware.com/products/vsphere-hypervisor/overview.html`.

[61] WALDSPURGER, C. A. Memory resource management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (New York, NY, USA, 2002), OSDI '02, ACM, pp. 181–194.

[62] WALDSPURGER, C. A., VENKATASUBRAMANIAN, R., GARTHWAITE, A. T., AND BASKAKOV, Y. Efficient online construction of miss rate curves, Apr. 2014. U.S. Patent #8,694,728. Filed Nov. 2010. Assigned to VMware, Inc.

[63] WEST, R., ZAROO, P., WALDSPURGER, C. A., AND ZHANG, X. Online cache modeling for commodity multicore processors. *SIGOPS Oper. Syst. Rev. 44*, 4 (Dec. 2010), 19–29.

[64] WIRES, J. Private communication, Jan 2015.

[65] WIRES, J., INGRAM, S., DRUDI, Z., HARVEY, N. J. A., AND WARFIELD, A. Characterizing storage workloads with counter stacks. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2014), OSDI'14, USENIX Association, pp. 335–349.

[66] XIE, F., CONDICT, M., AND SHETE, S. Estimating duplication by content-based sampling. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)* (San Jose, CA, 2013), USENIX, pp. 181–186.

[67] YANG, T., BERGER, E. D., KAPLAN, S. F., AND MOSS, J. E. B. CRAMM: Virtual memory support for garbage-collected applications. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 103–116.

[68] Yang, T., Hertz, M., Berger, E. D., Kaplan, S. F., and Moss, J. E. B. Automatic heap sizing: Taking real memory into account. In *Proceedings of the 4th International Symposium on Memory Management* (New York, NY, USA, 2004), ISMM '04, ACM, pp. 61–72.

[69] Zhao, W., Jin, X., Wang, Z., Wang, X., Luo, Y., and Li, X. Low cost working set size tracking. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2011), USENIXATC'11, USENIX Association, pp. 17–17.

[70] Zhao, W., and Wang, Z. Dynamic memory balancing for virtual machines. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2009), VEE '09, ACM, pp. 21–30.

[71] Zhong, Y., and Chang, W. Sampling-based program locality approximation. In *Proceedings of the 7th International Symposium on Memory Management* (New York, NY, USA, 2008), ISMM '08, ACM, pp. 91–100.

[72] Zhou, P., Pandey, V., Sundaresan, J., Raghuraman, A., Zhou, Y., and Kumar, S. Dynamic tracking of page miss ratio curve for memory management. *SIGOPS Oper. Syst. Rev. 38*, 5 (Oct. 2004), 177–188.

ANViL: Advanced Virtualization for Modern Non-Volatile Memory Devices

Zev Weiss*, Sriram Subramanian†, Swaminathan Sundararaman†,
Nisha Talagala†, Andrea C. Arpaci-Dusseau*, Remzi H. Arpaci-Dusseau*
* *University of Wisconsin-Madison*, † *SanDisk, Inc.*

Abstract

We present a new form of storage virtualization based on block-level address remapping. By allowing the host system to manipulate this address map with a set of three simple operations (clone, move, and delete), we enable a variety of useful features and optimizations to be readily implemented, including snapshots, deduplication, and single-write journaling. We present a prototype implementation called Project ANViL and demonstrate its utility with a set of case studies.

1 Introduction

Virtualization has been widely employed as a technique for managing and exploiting the available resources in computing systems, from memory and processors to entire machines [1,2,4,5,9,22]. Virtual memory in particular has enabled numerous features and optimizations, including the `mmap(2)` interface to file I/O, shared libraries, efficient `fork(2)`, zero-copy I/O, and page sharing between virtual machines [3, 29].

Storage virtualization, however, while conceptually similar to memory virtualization, has typically been of limited use to applications, focusing instead on storage management by introducing abstraction between physical storage layout and the logical device as presented to a host or application using it [10, 13, 28]. Features and functionality enabled by storage virtualization, such as deduplication, replication, and thin-provisioning, remain hidden behind the block device interface. While highly useful, the features of existing storage virtualization systems are primarily limited to administrative functionality, such as defining and provisioning LUNs, offering nothing to actual applications beyond standard read and write operations. As others have shown, these limitations in storage virtualization result in sub-optimal application performance and duplication of functionality across different layers in the storage stack [8, 11, 18, 21].

Some of the limits of storage virtualization have been addressed in recent research on Flash Translation Layers (FTLs), with new machinery proposed to support Atomic Writes, Persistent TRIM, and Sparseness [17, 18, 20, 21, 24]. These extensions enable applications to better leverage the power of virtualization already built into the FTL and also extensions enable the removal of redundant functionality across system layers, resulting in better flash endurance and application-level performance [16, 21].

We propose a simple yet powerful set of primitives based on *fine-grained address remapping* at both the block and extent level. As we will show, fine-grained address remapping provides the flexibility needed to benefit applications while still retaining the generality necessary to provide the functionality offered by existing virtualized volume managers. By allowing the host to manipulate the block-level logical-to-physical address map with *clone*, *move*, and *delete* operations, we enable storage virtualization to more closely resemble virtualized memory in its fine-grained flexibility and broad utility, though in a manner adapted to the needs of persistent storage.

We illustrate the utility of our approach by developing the Advanced Nonvolatile-memory Virtualization Layer (ANViL), a prototype implementation of fine-grained address remapping as a stacking block device driver, to efficiently implement both file and volume snapshots, deduplication, and single-write journaling. More specifically, we demonstrate how ANViL can provide high performance volume snapshots, offering as much as a 7× performance improvement over an existing copy-on-write implementation of this feature. We show how ANViL can be used to allow common, conventional file systems to easily add support for file-level snapshots without requiring any radical redesign. We also demonstrate how it can be leveraged to provide a performance boost of up to 50% for transactional commits in a journaling file system.

The remainder of this paper is organized as follows: we begin by describing how ANViL fits in naturally in the context of modern flash devices (§2) and detailing the extended device interface we propose (§3). We then discuss the implementation of ANViL (§4), describe a set of case studies illustrating a variety of useful real-world applications (§5), and conclude (§6).

2 Background

Existing storage virtualization systems focus their feature sets primarily on functionality "behind" the block interface, offering features like replication, thin-provisioning, and volume snapshots geared toward simplified and improved storage administration [10, 28]. They offer little, however, in the way of added functionality to the *consumers* of the block interface: the file systems, databases, and other applications that actually access data from the virtualized storage. Existing storage technologies, particularly those found in increasingly-popular flash devices, offer much of the infrastructure necessary to provide more advanced storage virtualization that could provide a richer interface directly beneficial to applications.

At its innermost physical level, flash storage does not offer the simple read/write interface of conventional hard disk drives (HDDs), around which existing storage software has been designed. While reads can be performed simply, a write (or *program*) operation must be preceded by a relatively slow and energy-intensive *erase* operation on a larger erase block (often hundreds of kilobytes or larger), before which any live data in the erase block must be copied elsewhere. Flash storage devices typically employ a *flash translation layer* (FTL) to simplify integration of this more complex interface into existing systems by adapting the native flash interface to the simpler HDD-style read/write interface, hiding the complexity of program/erase cycles from other system components and making the flash device appear essentially as a faster HDD. In order to achieve this, FTLs typically employ log-style writing, in which data is never overwritten in-place, but instead appended to the head of a log [23]. The FTL then maintains an internal address-remapping table to track which locations in the physical log correspond to which addresses in the logical block address space provided to higher layers of the storage stack [12, 26].

Such an address map provides most of the machinery that would be necessary to provide more sophisticated storage virtualization, but its existence is not exposed to the host system, preventing its capabilities from being fully exploited. A variety of primitives have been proposed to better expose the internal power of flash translation layers and similar log and remapping style systems, including atomic writes, sparseness (thin provisioning), Persistent TRIM, and cache-friendly garbage collection models [18, 20, 21, 24, 30]. These have been shown to have value for a range of applications from file systems to databases, key-value stores, and caches.

3 Interfaces

Address-remapping structures exist in FTLs and storage engines that provide thin provisioning and other storage virtualization functions today. We propose an extended block interface that enables a new form of storage virtualization by introducing three operations to allow the host system to directly manipulate such an address map.

3.1 Operations

Range Clone: `clone(src, len, dst)`: The range clone operation instantiates new mappings in a given range of logical address space (the *destination* range) that point to the same physical addresses mapped at the corresponding logical addresses in another range (the *source* range); upon completion the two ranges share storage space. A read of an address in one range will return the same data as would be returned by a read of the corresponding address in the other range. This operation can be used to quickly relocate data from one location to another without incurring the time, space, and I/O bandwidth costs of a simplistic read-and-rewrite copy operation.

Range Move: `move(src, len, dst)`: The range move operation is similar to a range clone, but leaves the source logical address range unmapped. This operation has the effect of efficiently transferring data from one location to another, again avoiding the overheads of reading in data and writing it back out to a new location.

Range Delete: `delete(src, len)`: The range delete operation simply unmaps a range of the logical address space, effectively deleting whatever data had been present there. This operation is similar to the `TRIM` or `DISCARD` operation offered by existing SSDs. However, unlike `TRIM` or `DISCARD`, which are merely advisory, the stricter range delete operation guarantees that upon acknowledgment of completion the specified logical address range is persistently unmapped. Range deletion is conceptually similar to the Persistent TRIM operation defined in prior work [15, 20]. Our work extends previous concepts by combining this primitive with the above clone and move operations for additional utility.

Under this model, a given logical address can be either mapped or unmapped. A read of a mapped address returns the data stored at the corresponding physical address. A read of an unmapped address simply returns a block of zeros. A write to a logical address, whether mapped or unmapped, allocates a new location in physical storage for the updated logical address. If the logical address previously shared physical space with one or more additional logical addresses, that mapping will be decoupled, with the affected logical address now pointing to a new physical location while the other logical addresses retain their original mapping.

3.2 Complementary Properties

While giving the host system the ability to manipulate the storage address map is of course the primary aim of our proposed interface, other properties complement our interfaces nicely and make them more useful in practice for real-world storage systems.

Sparseness or Thin Provisioning: In conventional storage devices, the logical space exposed to the host system is mapped one-to-one to the (advertised) physical capacity of the device. However, the existence of the range clone operation implies that the address map must be many-to-one. Thus, in order to retain the ability to utilize the available storage capacity, the logical address space must be expanded – in other words, the device must be *thin-provisioned* or *sparse*. The size of the logical address space, now decoupled from the physical capacity of the device, determines the upper limit on the total number of cloned mappings that may exist for a given block.

Durability: The effects of a range operation must be crash-safe in the same manner that an ordinary data write is: once acknowledged as complete, the alteration to the address map must persist across a crash or power loss. This requirement implies that the metadata modification must be synchronously persisted, and thus that each range operation implies a write to the underlying physical storage media.

Atomicity: Because it provides significant added utility for applications in implementing semantics such as transactional updates, we propose that a vector of range operations may be submitted as a single atomic batch, guaranteeing that after a crash or power loss, the effects of either *all* or *none* of the requested operations will remain persistent upon recovery. Log-structuring (see §4.1) makes this relatively simple to implement.

4 Implementation

In this section we describe the implementation of our prototype, the Advanced Nonvolatile-memory Virtualization Layer (ANViL), a Linux kernel module that acts as a generic stacking block device driver. ANViL runs on top of single storage devices as well as RAID arrays of multiple devices and is equally at home on either. It is not a full FTL, but it bears a strong resemblance to one. Though an implementation within the context of an existing FTL would have been a possibility, we chose instead to build ANViL as a separate layer to simplify development.

4.1 Log Structuring

In order to support the operations described earlier (§3), ANViL is implemented as a log-structured block device. Every range operation is represented by a note written to the log specifying the point in the logical ordering of updates at which it was performed. The note also records the alterations to the logical address map that were performed; this simplifies reconstruction of the device's metadata after a crash. Each incoming write is redirected to a new physical location, so updates to a given logical range do not affect other logical ranges which might share physical data. Space on the backing device is managed in large segments (128MB by default); each segment is written sequentially and a log is maintained that links the segments together in temporal order.

4.2 Metadata Persistence

Whenever ANViL receives a write request, before acknowledging completion it must store in non-volatile media not only the data requested to be written, but also any updates to its own internal metadata necessary to guarantee that it will be able to read the block back even after a crash or power loss. The additional metadata is small (24 bytes per write request, independent of size), but due to being a stacked layer of the block IO path, writing an additional 24 bytes would require it to write out another entire block. Done naïvely, the extra blocks would incur an immediate 100% write amplification for a workload consisting of single-block writes, harming both performance and flash device lifespan. However, for a workload with multiple outstanding write requests (a write IO queue depth greater than one), metadata updates for multiple requests can be batched together into a single block write, amortizing the metadata update cost across multiple writes.

ANViL thus uses an adaptive write batching algorithm, which, upon receiving a write request, waits for a small period of time to see if further write requests arrive, increasing the effectiveness of this metadata batching optimization, while balancing the time spent waiting for another write with impact on the latency of the current write.

4.3 Space Management

Space on the backing device is allocated at block granularity for incoming write requests. When a write overwrites a logical address that was already written and thus mapped to an existing backing-device address, the new write is allocated a new address on the backing device and the old mapping for the logical address is deleted and replaced by a mapping to the new backing device address. When no mappings to a given block of the backing device remain, that block becomes "dead" and its space may be reclaimed. However, in order to maintain large regions of space in the backing device so as to allow for sequential writing, freeing individual blocks as they become invalid is not a good approach for ANViL. Instead, the minimum unit of space reclamation is one segment.

A background garbage collector continuously searches for segments of backing device space that are underutilized (i.e. have a large number of invalid blocks). When such a segment is found, its remaining live blocks are copied into a new segment (appended at the current head of the log as with a normal write), any logical addresses mapped to them are updated to point to the new location they have been written out to, and finally the entire segment is returned to the space allocator for reuse.

5 Case Studies

Here we demonstrate the generality and utility of our range operations by implementing, with relatively little effort, a number of features useful to other components across a broad range of the storage stack, including volume managers (enabling simple and efficient volume snapshots), file systems (easily-integrated file snapshots), and transactional storage systems such as relational databases (allowing transactional updates without the double-write penalty). All experiments were performed on an HP DL380p Gen8 server with two six-core (12-thread) 2.5GHz Intel Xeon processors and a 785GB Fusion-io ioDrive2, running Linux 3.4.

5.1 Snapshots

Snapshots are an important feature of modern storage systems and have been implemented at different layers of the storage stack from file systems to block devices [25]. ANViL easily supports snapshots at multiple layers; here we demonstrate file- and volume-level snapshots.

5.1.1 File Snapshots

File-level snapshots enable applications to checkpoint the state of individual files at arbitrary points in time, but are only supported by a few recent file systems [7]. Many widely-used file systems, such as ext4 [19] and XFS [27], do not offer file-level snapshots, due to the significant design and implementation complexity required.

ANViL enables file systems to support file-level snapshots with minimal implementation effort and no changes to their internal data structures. Snapshotting individual files is simplified with range clones, as the file system has only to allocate logical address space and issue a range operation to clone the address mappings from the existing file into the newly-allocated address space [14].

With just a few hundred lines of code, we have added an `ioctl` to ext4 to allow a zero-copy implementation of the standard `cp` command, providing an efficient (in both space and time) file-snapshot operation. Figure 1 shows, for varying file sizes, the time taken to copy a file using the standard `cp` command on an ext4 file system mounted on an ANViL device in comparison to the time taken to copy the file using our special range-clone

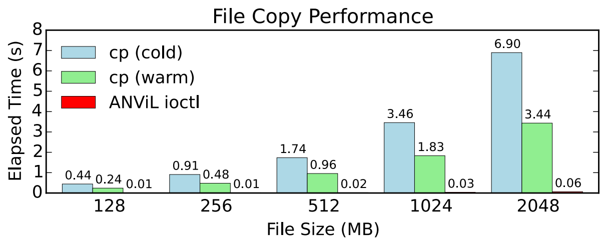


Figure 1: Time to copy files of various sizes via standard `cp` with both a cold and a warm page cache, and using a special ANViL `ioctl` in our modified version of ext4.

`ioctl`. Unsurprisingly, the range-clone based file copy is dramatically faster than the conventional read-and-write approach used by the unmodified `cp`, copying larger files in orders of magnitude less time. Also, unlike standard `cp`, the clone based implementation shares physical space between copies, making it vastly more storage efficient as normal for thinly provisioned snapshots.

5.1.2 Volume Snapshots

Volume snapshots are similar to file snapshots, but even simpler to implement. We merely identify the range of blocks that represent a volume and clone it into a new range of logical address space, which a volume manager can then provide access to as an independent volume.

Volume snapshots via range-clones offer much better performance than the snapshot facilities offered by some existing systems, such as Linux's built-in volume manager, LVM. LVM snapshots are (somewhat notoriously) slow, because they operate via copy-on-write of large extents of data (2MB by default) for each extent that is written to in the volume of which the snapshot was taken. To quantify this, we measure the performance of random writes at varying queue depths on an LVM volume and on ANViL, both with and without a recently-activated snapshot. In Figure 2, we see that while the LVM volume suffers a dramatic performance hit when a snapshot is active, ANViL sees little change in performance, since it instead uses its innate redirect-on-write mechanism.

5.2 Deduplication

Data deduplication is often employed to eliminate data redundancy and better utilize storage capacity by identifying pieces of identical data and collapsing them together to share the same physical space. Deduplication can of course be implemented easily using a range clone operation. As with snapshots, deduplication can be performed at different layers of the storage stack. Here we show how block-level deduplication can be easily supported by a file system running on top of an ANViL device.

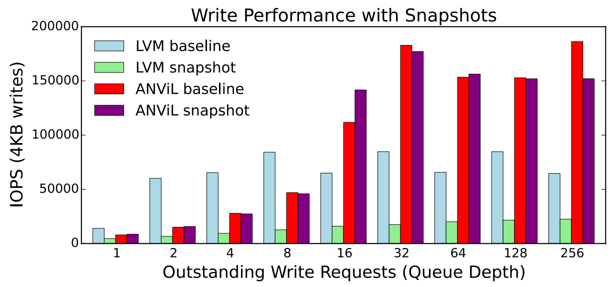


Figure 2: Random write IOPS on ANViL and LVM, both in isolation and with a recently-activated snapshot. The baseline bars illustrate ANViL's raw I/O performance. Its relatively low performance at small queue depths is due to the overhead incurred by its metadata updates.

Extending the same `ioctl` used to implement file snapshots (§5.1.1), we added an optional flag to specify that the file system should, as a single atomic operation, read the two indicated file ranges and then conditionally perform a range clone if and only if they contain identical data. This operation provides a base primitive that can be used as the underlying mechanism for a userspace deduplication tool, with the atomicity necessary to allow it to operate safely in the presence of possible concurrent file modifications. Without this locking it would risk losing data written to files in a time-of-check-to-time-of-use race between the deduplicator detecting that two block ranges are identical (the check) and performing the range-copy operation (the use). While the simplistic proof-of-concept deduplication system we have is unable to detect previously-deduplicated blocks and avoid re-processing them, the underlying mechanism could be employed by a more sophisticated offline deduplicator without this drawback (or even, with appropriate plumbing, an online one).

5.3 Single-Write Journaling

Journaling is widely used to provide atomicity to multi-block updates and thus ensure metadata (and sometimes data) consistency in systems such as databases and file systems. Such techniques are required because storage devices typically do not provide any atomicity guarantees beyond a single block write. Unfortunately, journaling causes each journaled update to be performed twice: once to the journal region and then to the final location of the data. In case of failure, updates that have been committed to the journal are replayed at recovery time, and uncommitted updates are discarded. ANViL, however, can leverage its redirect-on-write nature and internal metadata management to support a multi-block atomic write operation. With this capability, we can avoid the double-write penalty of journaling and thus improve both performance and the lifespan of the flash device.

By making a relatively small modification to a journaling file system, we can use a vectored atomic range move operation to achieve this optimization. When the file system would write the commit block for a journal transaction, it instead issues a single vector of range moves to atomically relocate all metadata (and/or data) blocks in the journal transaction to their "home" locations in the main file system. Figure 3 illustrates an atomic commit operation via range moves. This approach is similar to Choi et al.'s JFTL [6], though unlike JFTL the much more general framework provided by ANViL is not tailored specifically to journaling file systems.

Using range moves in this way obviates the need for a second write to copy each block to its primary location, since the range move has already put them there, eliminating the double-write penalty inherent to conventional journaling. This technique is equally applicable to metadata journaling and full data journaling; with the latter this means that a file system can achieve the stronger consistency properties offered by data journaling without paying the penalty of the doubling of write traffic incurred by journaling without range moves. By halving the amount of data written, flash device lifespan is also increased.

Commit-via-range-move also obviates the need for any journal recovery at mount time, since any transaction that has committed will need no further processing or IO, and any transaction in the journal that has not completed should not be replayed anyway (for consistency reasons). This simplification would allow the elimination of over 700 lines of (relatively intricate) recovery code from the jbd2 codebase.

In effect, this approach to atomicity simply exposes to the application (the file system, in this case) the internal operations necessary to stitch together a vectored atomic write operation from more primitive operations: the application writes its buffers to a region of scratch space (the journal), and then, once all of the writes have completed, issues a single vectored atomic range move to put each block in its desired location.

We have implemented single-write journaling in ext4's jbd2 journaling layer; it took approximately 100 lines of new code and allowed the removal of over 900 lines of existing commit and recovery code. Figure 4 shows the performance results for write throughput in data journaling mode of a process writing to a file in varying chunk sizes and calling `fdatasync` after each write. In all cases ext4a (our modified, ANViL-optimized version of ext4) achieves substantially higher throughput than the baseline ext4 file system. At small write sizes the relative gain of ext4a is larger, because in addition to eliminating the double-write of file data, the recovery-less

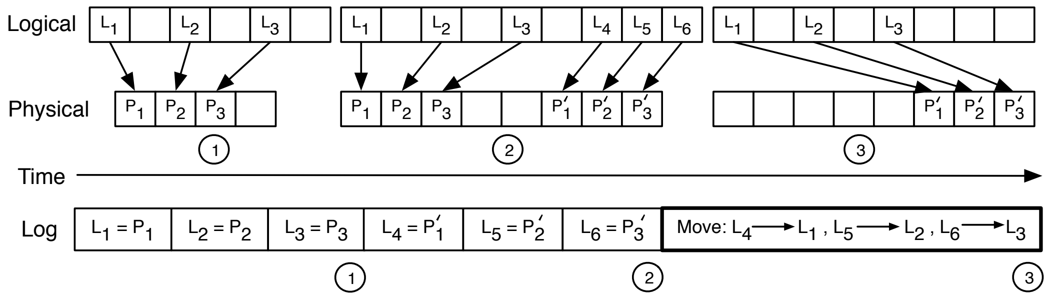


Figure 3: **Transactions via address remapping** *By using an application-managed scratch area, atomic transactional updates can be implemented using range operations. At ① the system is in its consistent pre-transaction state, with logical blocks L_1, L_2, and L_3 each mapped to blocks containing the initial versions of the relevant data. Between ① and ②, new versions of these blocks are written out and mapped to logical addresses in a temporary scratch area (L_4, L_5, and L_6). Note that this portion is not required to proceed atomically. Once the temporary locations have all been populated, an atomic range-move operation remaps the new blocks at L_4, L_5, and L_6 to L_1, L_2, and L_3, respectively, leading to ③, at which point the transaction is fully committed.*

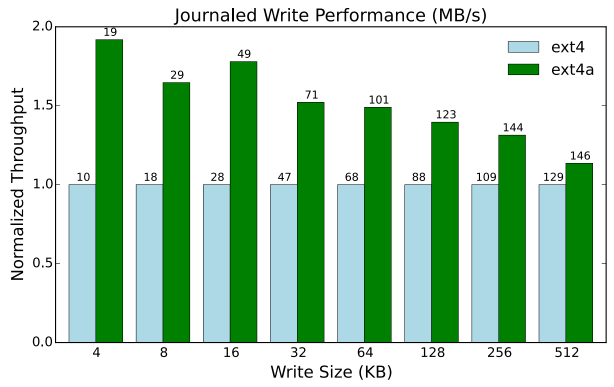


Figure 4: Data journaling write throughput with ANViL-optimized ext4a compared to unmodified ext4. Each bar is labeled with absolute write bandwidth.

nature of single-write journaling also obviates the need for writing the start and commit blocks of each journal transaction; for small transactions the savings from this are proportionally larger. At larger write sizes, the reason that the performance gain is less than the doubling that might be expected (due to halving the amount of data written) is that despite consisting purely of synchronous file writes, the workload is actually insufficiently IO-bound. The raw performance of the storage device is high enough that CPU activity in the file system consumes approximately 50% of the workload's execution time; jbd2's `kjournald` thread (which performs all journal writes) is incapable of keeping the device utilized, and its single-threadedness means that adding additional userspace IO threads to the workload does little to increase device IO bandwidth utilization. Adding a second thread to the 512KB write workload increases throughput from 132 MB/s to 140 MB/s; four threads actually *decreases* throughput to 128 MB/s.

The mechanism underlying single-write journaling could be more generally applied to most forms of write-ahead logging, such as that employed by relational database management systems [21].

6 Conclusions

The above case studies show that with a simple but powerful remapping mechanism, a single log structured storage layer can provide upstream software with both high performance and a flexible storage substrate.

Virtualization is an integral part of modern systems, and with the advent of flash it has become important to consider storage virtualization beyond volume management in order to uncover the true potential of the technology. In this paper we have proposed advanced storage virtualization with a set of interfaces giving applications fine-grained control over storage address remapping. Their implementation is a natural extension of common mechanisms present in log-structured datastores such as FTLs, and we demonstrated, with a set of practical case studies with our ANViL prototype, the utility and generality of this interface. Our work to date shows that the proposed interfaces have enough flexibility to provide a great deal of added utility to applications while remaining relatively simple to integrate.

7 Acknowledgments

We thank the anonymous reviewers, Tyler Harter, and our shepherd Jiri Schindler for their feedback, the rest of the ANViL development team, and Vijay Chidambaram for his assistance with our modifications to ext4.

References

[1] Keith Adams and Ole Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*, Seattle, Washington, March 2008.

[2] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, Pacific Grove, California, October 1991.

[3] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using KSM. In *Ottawa Linux Symposium*, 2009.

[4] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.80 edition, May 2014.

[5] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 143–156, Saint-Malo, France, October 1997.

[6] Hyun Jin Choi, Seung-Ho Lim, and Kyu Ho Park. JFTL: A Flash Translation Layer Based on a Journal Remapping for Flash Memory. *ACM Transactions on Storage (TOS)*, 4(4), February 2009.

[7] Chris Mason. Btrfs Design. `http://oss.oracle.com/projects/btrfs/dist/documentation/btrfs-design.html`, 2011.

[8] Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Bridging the Information Gap in Storage Protocol Stacks. In *Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, pages 177–190, Monterey, California, June 2002.

[9] Boris Dragovic, Keir Fraser, Steve Hand, Tim Harris, Alex Ho, Ian Pratt, Andrew Warfield, Paul Barham, and Rolf Neugebauer. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, New York, October 2003.

[10] 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 *Proceedings of the USENIX Annual Technical Conference (USENIX '08)*, Boston, Massachusetts, June 2008.

[11] Gregory R. Ganger. Blurring the Line Between OSes and Storage Devices. Technical Report CMU-CS-01-166, Carnegie Mellon University, December 2001.

[12] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. DFTL: a Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*, pages 229–240, Washington, DC, March 2009.

[13] Red Hat. LVM2 Resource Page. `http://www.sourceware.org/lvm2/`.

[14] Dave Hitz, James Lau, and Michael Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '94)*, San Francisco, California, January 1994.

[15] Nikolaus Jeremic, Gero Mühl, Anselm Busse, and Jan Richling. Enabling TRIM Support in SSD RAIDs. Technical report, Technical Report Informatik Preprint CS-05-11, Department of Computer Science, University of Rostock, 2011.

[16] William K. Josephson, Lars A. Bongo, Kai Li, and David Flynn. DFS: A File System for Virtualized Flash Storage. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST '10)*, San Jose, California, February 2010.

[17] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed Virtual Disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, Cambridge, Massachusetts, October 1996.

[18] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, Raju Rangaswami, Sushma Devendrappa, Bharath Ramsundar, and Sriram Ganesan.

NVMKV: A Scalable and Lightweight Flash Aware Key-Value Store. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, Philadelphia, PA, June 2014. USENIX Association.

[19] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The New Ext4 Filesystem: Current Status and Future Plans. In *Ottawa Linux Symposium (OLS '07)*, Ottawa, Canada, July 2007.

[20] David Nellans, Michael Zappe, Jens Axboe, and David Flynn. ptrim() + exists(): Exposing New FTL Primitives to Applications. In *Proceedings of the Non-Volatile Memory Workshop*, NVMW '11, 2011.

[21] Xiangyong Ouyang, David W. Nellans, Robert Wipfel, David Flynn, and Dhabaleswar K. Panda. Beyond Block I/O: Rethinking Traditional Storage Primitives. In *HPCA*, pages 301–311. IEEE Computer Society, 2011.

[22] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 31–39, Palo Alto, California, 1991.

[23] Mendel Rosenblum and John Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[24] Mohit Saxena, Michael M. Swift, and Yiying Zhang. FlashTier: A Lightweight, Consistent and Durable Storage Cache. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 267–280, New York, NY, USA, 2012. ACM.

[25] Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Snapshots in a Flash with ioSnap. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, 2014.

[26] Kyoungmoon Sun, Seungjae Baek, Jongmoo Choi, Donghee Lee, Sam H. Noh, and Sang Lyul Min. LTFTL: Lightweight Time-shift Flash Translation Layer for Flash Memory Based Embedded Storage. In *Proceedings of the 8th ACM International Conference on Embedded Software*, EMSOFT '08, 2008.

[27] Adan Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '96)*, San Diego, California, January 1996.

[28] Veritas. Features of VERITAS Volume Manager for Unix and VERITAS File System. `http://www.veritas.com/us/products/volumemanager/whitepaper-02.html`, July 2005.

[29] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[30] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, Swaminathan Sundararaman, and Robert Wood. HEC: Improving Endurance of High Performance Flash-based Cache Devices. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, 2013.

Reducing File System Tail Latencies with *Chopper*

Jun He, Duy Nguyen[†], Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
Department of Computer Sciences, [†]Department of Statistics
University of Wisconsin–Madison

Abstract

We present *Chopper*, a tool that efficiently explores the vast input space of file system policies to find behaviors that lead to costly performance problems. We focus specifically on block allocation, as unexpected poor layouts can lead to high tail latencies. Our approach utilizes sophisticated statistical methodologies, based on Latin Hypercube Sampling (LHS) and sensitivity analysis, to explore the search space efficiently and diagnose intricate design problems. We apply *Chopper* to study the overall behavior of two file systems, and to study Linux ext4 in depth. We identify four internal design issues in the block allocator of ext4 which form a large tail in the distribution of layout quality. By removing the underlying problems in the code, we cut the size of the tail by an order of magnitude, producing consistent and satisfactory file layouts that reduce data access latencies.

1 Introduction

As the distributed systems that power the cloud have matured, a new performance focus has come into play: *tail latency*. As Dean and Barroso describe, long tails can dramatically harm interactive performance and thus limit the applications that can be effectively deployed at scale in modern cloud-based services [17]. As a result, a great deal of recent research effort has attacked tail latency directly [6,49,51]; for example, Alizadeh et al. show how to reduce the network latency of the 99th percentile by a factor of ten through a combination of novel techniques [6].

The fundamental reason that reducing such tail latency is challenging is that rare, corner-case behaviors, which have little impact on a single system, can dominate when running a system at scale [17]. Thus, while the well-tested and frequently-exercised portions of a system perform well, the unusual behaviors that are readily ignored on one machine become the common case upon one thousand (or more) machines.

To build the next generation of robust, predictably performing systems, we need an approach that can readily discover corner-case behaviors, thus enabling a developer to find and fix intrinsic tail-latency problems before deployment. Unfortunately, finding unusual behavior is hard: just like exploring an infinite state space for correctness bugs remains an issue for today's model checkers [10, 19], discovering the poorly-performing tail-influencing behaviors presents a significant challenge.

One critical contributor to tail latency is the local file system [8]. Found at the heart of most distributed file systems [20, 47], local file systems such as Linux ext4, XFS, and btrfs serve as the building block for modern scalable storage. Thus, if rare-case performance of the local file system is poor, the performance of the distributed file system built on top of it will suffer.

In this paper, we present *Chopper*, a tool that enables developers to discover (and subsequently repair) high-latency operations within local file systems. *Chopper* currently focuses on a critical contributor to unusual behavior in modern systems: block allocation, which can reduce file system performance by one or more orders of magnitude on both hard disk and solid state drives [1, 11, 13, 30, 36]. With *Chopper*, we show how to find such poor behaviors, and then how to fix them (usually through simple file-system repairs).

The key and most novel aspect of *Chopper* is its usage of advanced statistical techniques to search and investigate an infinite performance space systematically. Specifically, we use Latin hypercube sampling [29] and sensitivity analysis [40], which has been proven efficient in the investigation of many-factor systems in other applications [24,31,39]. We show how to apply such advanced techniques to the domain of file-system performance analysis, and in doing so make finding tail behavior tractable.

We use *Chopper* to analyze the allocation performance of Linux ext4 and XFS, and then delve into a detailed analysis of ext4 as its behavior is more complex and varied. We find four subtle flaws in ext4, including behaviors that spread sequentially-written files over the entire disk volume, greatly increasing fragmentation and inducing large latency when the data is later accessed. We also show how simple fixes can remedy these problems, resulting in an order-of-magnitude improvement in the tail layout quality of the block allocator. *Chopper* and the ext4 patches are publicly available at:

`research.cs.wisc.edu/adsl/Software/chopper`

The rest of the paper is organized as follows. Section 2 introduces the experimental methodology and implementation of *Chopper*. In Section 3, we evaluate ext4 and XFS as black boxes and then go further to explore ext4 as a white box since ext4 has a much larger tail than XFS. We present detailed analysis and fixes for internal allocator design issues of ext4. Section 4 introduces related work. Section 5 concludes this paper.

2 Diagnosis Methodology

We now describe our methodology for discovering interesting tail behaviors in file system performance, particularly as related to block allocation. The file system input space is vast, and thus cannot be explored exhaustively; we thus treat each file system experiment as a simulation, and apply a sophisticated sampling technique to ensure that the large input space is explored carefully.

In this section, we first describe our general experimental approach, the inputs we use, and the output metric of choice. We conclude by presenting our implementation.

2.1 Experimental Framework

The Monte Carlo method is a process of exploring simulation by obtaining numeric results through repeated random sampling of inputs [38,40,43]. Here, we treat the file system itself as a simulator, thus placing it into the Monte Carlo framework. Each run of the file system, given a set of inputs, produces a single output, and we use this framework to explore the file system as a black box.

Each input factor X_i $(i = 1, 2, ..., K)$ (described further in Section 2.2) is estimated to follow a distribution. For example, if small files are of particular interest, one can utilize a distribution that skews toward small file sizes. In the experiments of this paper, we use a uniform distribution for fair searching. For each factor X_i, we draw a sample from its distribution and get a vector of values $(X_i^1, X_i^2, X_i^3, .., X_i^N)$. Collecting samples of all the factors, we obtain a matrix M.

$$M = \begin{bmatrix} X_1^1 & X_2^1 & ... & X_K^1 \\ X_1^2 & X_2^2 & ... & X_K^2 \\ ... & & & \\ X_1^N & X_2^N & ... & X_K^N \end{bmatrix} \qquad Y = \begin{bmatrix} Y^1 \\ Y^2 \\ ... \\ Y^N \end{bmatrix}$$

Each row in M, i.e., a *treatment*, is a vector to be used as input of one run, which produces one row in vector Y. In our experiment, M consists of columns such as the size of the file system and how much of it is currently in use. Y is a vector of the output metric; as described below, we use a metric that captures how much a file is spread out over the disk called *d-span*. M and Y are used for exploratory data analysis.

The framework described above allows us to explore file systems over different combinations of values for uncertain inputs. This is valuable for file system studies where the access patterns are uncertain. With the framework, block allocator designers can explore the consequences of design decisions and users can examine the allocator for their workload.

In the experiment framework, M is a set of treatments we would like to test, which is called an *experimental plan* (or *experimental design*). With a large input space, it is essential to pick input values of each factor and organize them in a way to efficiently explore the space in a limited number of runs. For example, even with our refined space in Table 1 (introduced in detail later), there are about 8×10^9 combinations to explore. With an overly optimistic speed of one treatment per second, it still would take 250 compute-years to finish just one such exploration.

Latin Hypercube Sampling (LHS) is a sampling method that efficiently explores many-factor systems with a large input space and helps discover surprising behaviors [25, 29, 40]. A *Latin hypercube* is a generalization of a *Latin square*, which is a square grid with only one sample point in each row and each column, to an arbitrary number of dimensions [12]. LHS is very effective in examining the influence of each factor when the number of runs in the experiment is much larger than the number of factors. It aids visual analysis as it exercises the system over the entire range of each input factor and ensures all levels of it are explored evenly [38]. LHS can effectively discover which factors and which combinations of factors have a large influence on the response. A poor sampling method, such as a completely random one, could have input points clustered in the input space, leaving large unexplored gaps in-between [38]. Our experimental plan, based on LHS, contains 16384 runs, large enough to discover subtle behaviors but not so large as to require an impractical amount of time.

2.2 Factors to Explore

File systems are complex. It is virtually impossible to study all possible factors influencing performance. For example, the various file system formatting and mounting options alone yield a large number of combinations. In addition, the run-time environment is complex; for example, file system data is often buffered in OS page caches in memory, and differences in memory size can dramatically change file system behavior.

In this study, we choose to focus on a subset of factors that we believe are most relevant to allocation behavior. As we will see, these factors are broad enough to discover interesting performance oddities; they are also not so broad as to make a thorough exploration intractable.

There are three categories of input factors in *Chopper*. The first category of factors describes the initial state of the file system. The second category includes a relevant OS state. The third category includes factors describing the workload itself. All factors are picked to reveal potentially interesting design issues. In the rest of this paper, a value picked for a factor is called a *level*. A set of levels, each of which is selected for a factor, is called a *treatment*. One execution of a treatment is called a *run*. We picked twelve factors, which are summarized in Table 1 and introduced as follows.

We create a virtual disk of **DiskSize** bytes, because block allocators may have different space management policies for disks of different sizes.

The **UsedRatio** factor describes the ratio of disk that

	Factor	Description	Presented Space
FS	DiskSize	Size of disk the file system is mounted on.	1,2,4,...,64GB
	UsedRatio	Ratio of used disk.	0, 0.2, 0.4, 0.6
	FreeSpaceLayout	Small number indicates high fragmentation.	1,2,...,6
OS	CPUCount	Number of CPUs available.	1,2
Workload	FileSize	Size of file.	8,16,24,...,256KB
	ChunkCount	Number of chunks each file is evenly divided into.	4
	InternalDensity	Degree of sparseness or overwriting.	0.2,0.4,...,2.0
	ChunkOrder	Order of writing the chunks.	permutation(0,1,2,3)
	Fsync	Pattern of `fsync()`.	****, *=0 or 1
	Sync	Pattern of `close()`, `sync()`, and `open()`.	***1, *=0 or 1
	FileCount	Number of files to be written.	1,2
	DirectorySpan	Distance of files in the directory tree.	1,2,3,...,12

Table 1: Factors in Experiment.

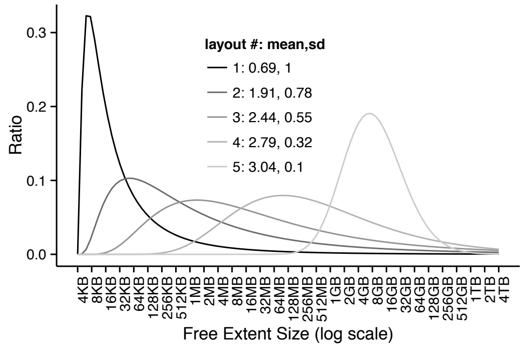


Figure 1: LayoutNumber. Degree of fragmentation represented as lognormal distribution.

has been used. *Chopper* includes it because block allocators may allocate blocks differently when the availability of free space is different.

The **FreeSpaceLayout** factor describes the contiguity of free space on disk. Obtaining satisfactory layouts despite a paucity of free space, which often arises when file systems are aged, is an important task for block allocators. Because enumerating all fragmentation states is impossible, we use six numbers to represent degrees from extremely fragmented to generally contiguous. We use the distribution of free extent sizes to describe the degree of fragmentations; the extent sizes follow lognormal distributions. Distributions of layout 1 to 5 are shown in Figure 1. For example, if layout is number 2, about $0.1 \times DiskSize \times (1 - UsedRatio)$ bytes will consist of 32KB extents, which are placed randomly in the free space. Layout 6 is not manually fragmented, in order to have the most contiguous free extents possible.

The **CPUCount** factor controls the number of CPUs the OS runs on. It can be used to discover scalability issues of block allocators.

The **FileSize** factor represents the size of the file to be written, as allocators may behave differently when different sized files are allocated. For simplicity, if there is more than one file in a treatment, all of them have the same size.

A chunk is the data written by a `write()` call. A file is often not written by only one call, but a series of writes. Thus, it is interesting to see how block allocators act with different numbers of chunks, which **ChunkCount** factor captures. In our experiments, a file is divided into multiple chunks of equal sizes. They are named by their positions in file, e.g., if there are four chunks, chunk-0 is at the head of the file and chunk-3 is at the end.

Sparse files, such as virtual machine images [26], are commonly-used and important. Files written non-sequentially are sparse at some point in their life, although the final state is not. On the other hand, overwriting is also common and can have effect if any copy-on-write strategy is adopted [34]. The **InternalDensity** factor describes the degree of coverage (e.g. sparseness or overwriting) of a file. For example, if InternalDensity is 0.2 and chunk size is 10KB, only the 2KB at the end of each chunk will be written. If InternalDensity is 1.2, there will be two writes for each chunk; the first write of this chunk will be 10KB and the second one will be 2KB at the end of the chunk.

The **ChunkOrder** factor defines the order in which the chunks are written. It explores sequential and random write patterns, but with more control. For example, if a file has four chunks, ChunkOrder=0123 specifies that the file is written from the beginning to the end; ChunkOrder=3210 specifies that the file is written backwards.

The **Fsync** factor is defined as a bitmap describing whether *Chopper* performs an `fsync()` call after each chunk is written. Applications, such as databases, often use `fsync()` to force data durability immediately [15, 23]. This factor explores how `fsync()` may interplay with allocator features (e.g., delayed allocation in Linux ext4 [28]). In the experiment, if ChunkOrder=1230 and Fsync=1100, *Chopper* will perform an `fsync()` after chunk-1 and chunk-2 are written, but not otherwise.

The **Sync** factor defines how we open, close, or sync the file system with each write. For example, if ChunkOrder=1230 and Sync=0011, *Chopper* will perform the three calls after chunk-3 and perform `close()` and `sync()` after chunk-0; `open()` is not called after the last chunk is written. All Sync bitmaps end with 1, in order to place data on disk before we inquire about layout information. *Chopper* performs `fsync()` before `sync()` if they both are requested for a chunk.

The **FileCount** factor describes the number of files written, which is used to explore how block allocators preserve spatial locality for one file and for multiple files. In the experiment, if there is more than one file, the chunks of each file will be written in an interleaved fashion. The ChunkOrder, Fsync, and Sync for all the files in a single treatment are identical.

Chopper places files in different nodes of a directory tree to study how parent directories can affect the data layouts. The **DirectorySpan** factor describes the distance between parent directories of the first and last files in a breadth-first traversal of the tree. If FileCount=1, DirectorySpan is the index of the parent directory in the breadth-first traversal sequence. If FileCount=2, the first file will be placed in the first directory, and the second one will be at the *DirectorySpan*-th position of the traversal sequence.

In summary, the input space of the experiments presented in this paper is described in Table 1. The choice is based on efficiency and simplicity. For example, we study relatively small file sizes because past studies of file systems indicates most files are relatively small [5, 9, 35]. Specifically, Agrawal et. al. found that over 90% of the files are below 256 KB across a wide range of systems [5]. Our results reveal many interesting behaviors, many of which also apply to larger files. In addition, we study relatively small disk sizes as large ones slow down experiments and prevent broad explorations in limited time. The file system problems we found with small disk sizes are also present with large disks.

Simplicity is also critical. For example, we use at most two files in these experiments. Writing to just two files, we have found, can reveal interesting nuances in block allocation. Exploring more files make the results more challenging to interpret. We leave further exploration of the file system input space to future work.

2.3 Layout Diagnosis Response

To diagnose block allocators, which aim to place data compactly to avoid time-consuming seeking on HDDs [7, 36] and garbage collections on SSDs [11, 30], we need an intuitive metric reflecting data layout quality. To this end, we define *d-span*, the distance in bytes between the first and last physical block of a file. In other words, *d-span* measures the worst allocation decision the allocator makes in terms of spreading data. As desired, *d-span* is an *indirect* performance metric, and, more importantly, an intuitive diagnostic signal that helps us find unexpected file-system behaviors. These behaviors may produce poor layouts that eventually induce long data access latencies. *d-span* captures subtle problematic behaviors which would be hidden if end-to-end performance metrics were used. Ideally, *d-span* should be the same size as the file.

d-span is not intended to be an one-size-fits-all metric. Being simple, it has its weaknesses. For example, it cannot distinguish cases that have the same span but different internal layouts. An alternative of *d-span* that we have investigated is to model data blocks as vertices in a graph and use *average path length* [18] as the metric. The minimum distance between two vertices in the graph is their corresponding distance on disk. Although this metric is able to distinguish between various internal layouts, we have found that it is often confusing. In contrast, *d-span* contains less information but is much easier to interpret.

In addition to the metrics above, we have also explored metrics such as number of data extents, layout score (fraction of contiguous blocks) [42], and normalized versions of each metric (e.g. $d\text{-}span/ideal\ d\text{-}span$). One can even create a metric by plugging in a disk model to measure quality. Our diagnostic framework works with all of these metrics, each of which allows us to view the system from a different angle. However, *d-span* has the best trade-off between information gain and simplicity.

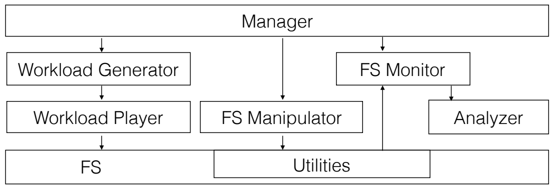


Figure 2: *Chopper* components.

2.4 Implementation

The components of *Chopper* are presented in Figure 2. The **Manager** builds an experimental plan and conducts the plan using the other components. The **FS Manipulator** prepares the file system for subsequent workloads. In order to speed up the experiments, the file system is mounted on an in-memory virtual disk, which is implemented as a loop-back device backed by a file in a RAM file system. The initial disk images are re-used whenever needed, thus speeding up experimentation and providing reproducibility. After the image is ready, the **Workload Generator** produces a workload description, which is then fed into the **Workload Player** for running. After playing the workload, the Manager informs the **FS Monitor**, which invokes existing system utilities, such as *debugfs* and *xfs_db*, to collect layout information. No kernel changes are needed. Finally, layout information is merged with workload and system information and fed into the **Analyzer**. The experiment runs can be executed in parallel to significantly reduce time.

3 The Tale of Tail

We use *Chopper* to help understand the policies of file system block allocators, to achieve more predictable and consistent data layouts, and to reduce the chances of performance fluctuations. In this paper, we focus on Linux ext4 [28] and XFS [41], which are among the most popular local file systems [2–4, 33].

For each file system, we begin in Section 3.1 by asking whether or not it provides robust file layout in the presence of uncertain workloads. If the file system is robust (i.e., XFS), then we claim success; however, if it is not (i.e., ext4), then we delve further into understanding the workload and environment factors that cause the unpredictable layouts. Once we understand the combination of factors that are problematic, in Section 3.2, we search for the responsible policies in the file system source code and improve those policies.

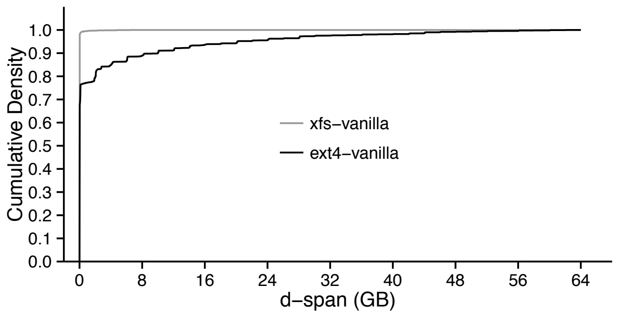


Figure 3: *d-span* CDFs of ext4 and XFS. The 90th%, 95th%, and max *d-span*s of ext4 are 10GB, 20GB, and 63GB, respectively. The 90th%, 95th%, and max *d-span*s of XFS are 2MB, 4MB, and 6GB, respectively.

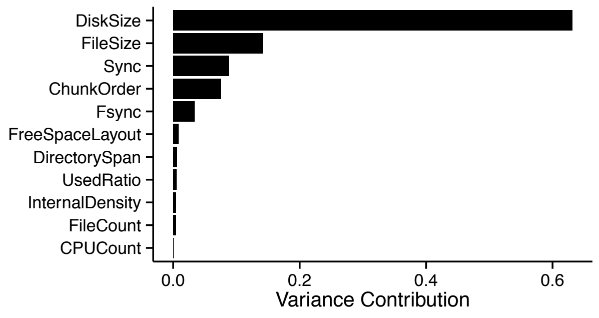


Figure 4: Contribution to *d-span* variance. It shows contributions calculated by factor prioritization of sensitivity analysis.

3.1 File System as a Black Box

3.1.1 Does a Tail Exist?

The first question we ask is whether or not the file allocation policies in Linux ext4 and XFS are robust to the input space introduced in Table 1.

To find out if there are tails in the resulting allocations, we conducted experiments with 16384 runs using *Chopper*. The experiments were conducted on a cluster of nodes with 16 GB RAM and two Opteron-242 CPUs [21]. The nodes ran Linux v3.12.5. Exploiting *Chopper*'s parallelism and optimizations, one full experiment on each file system took about 30 minutes with 32 nodes.

Figure 3 presents the empirical CDF of the resulting *d-span*s for each file system over all the runs; in runs with multiple files, the reported *d-span* is the maximum *d-span* of the allocated files. A large *d-span* value indicates a file with poor locality. Note that the file sizes are never larger than 256KB, so *d-span* with optimal allocation would be only 256KB as well.

The figure shows that the CDF line for XFS is nearly vertical; thus, XFS allocates files with relatively little variation in the *d-span* metric, even with widely differing workloads and environmental factors. While XFS may not be ideal, this CDF (as well as further experiments not shown due to space constraints) indicates that its block allocation policy is relatively robust.

In contrast, the CDF for ext4 has a significant tail. Specifically, 10% of the runs in ext4 have at least one file spreading over 10GB. This tail indicates instability in the ext4 block allocation policy that could produce poor layouts inducing long access latencies.

3.1.2 Which factors contribute to the tail?

We next investigate which workload and environment factors contribute most to the variation seen in ext4 layout. Understanding these factors is important for two reasons. First, it can help file system users to see which workloads run best on a given file system and to avoid those which do not run well; second, it can help file system developers track down the source of internal policy problems.

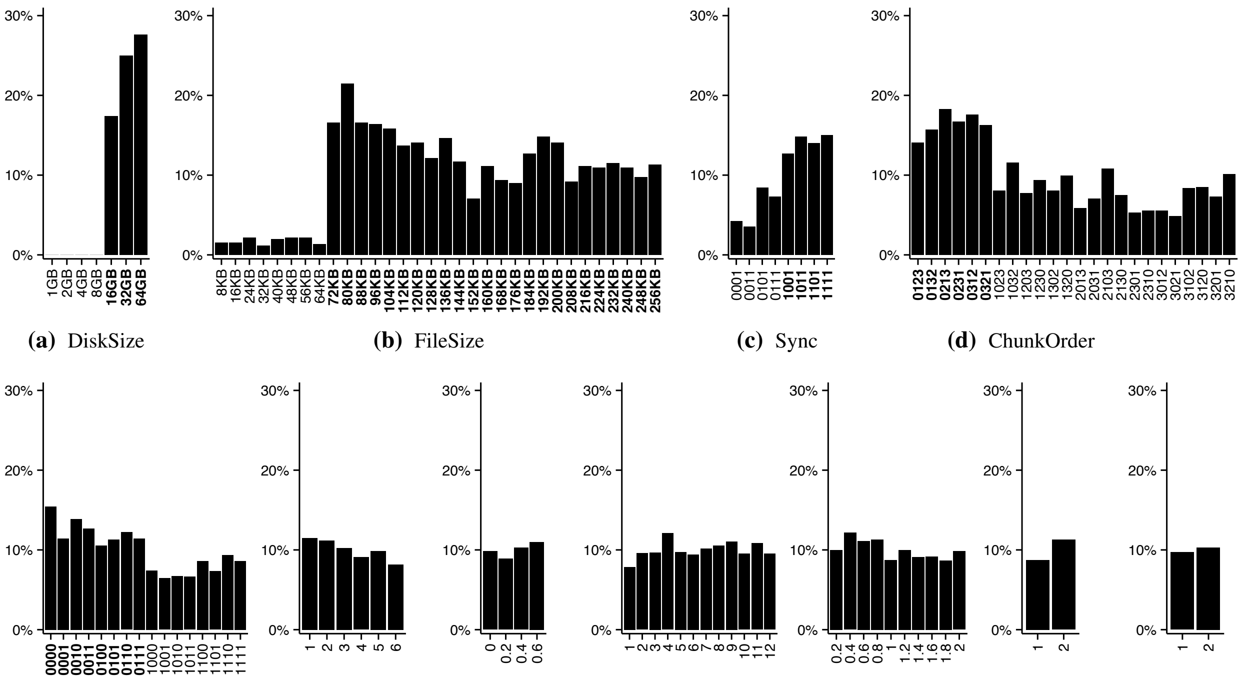
The contribution of a factor to variation can be calculated by variance-based *factor prioritization*, a technique in sensitivity analysis [38]. Specifically, the contribution of factor X_i is calculated by:

$$S_i = \frac{V_{X_i}(E_{X_{\sim i}}(Y|X_i = x_i^*))}{V(Y)}$$

S_i is always smaller than 1 and reports the ratio of the contribution by factor X_i to the overall variation. In more detail, if factor X_i is fixed at a particular level x_i^*, then $E_{X_{\sim i}}(Y|X_i = x_i^*)$ is the resulting mean of response values for that level, $V_{X_i}(E_{X_{\sim i}}(Y|X_i = x_i^*))$ is the variance among level means of X_i, and $V(Y)$ is the variance of all response values for an experiment. Intuitively, S_i indicates how much changing a factor can affect the response.

Figure 4 presents the contribution of each factor for ext4; again, the metric indicates the contribution of each factor to the variation of *d-span* in the experiment. The figure shows that the most significant factors are DiskSize, FileSize, Sync, ChunkOrder, and Fsync; that is, changing any one of those factors may significantly affect *d-span* and layout quality. DiskSize is the most sensitive factor, indicating that ext4 does not have stable layout quality with different disk sizes. It is not surprising that FileSize affects *d-span* considering that the definition *d-span* depends on the size of the file; however, the variance contributed by FileSize ($0.14 \times V(dspan_{real}) = 3 \times 10^{18}$) is much larger than ideally expected ($V(dspan_{ideal}) = 6 \times 10^{10}, dspan_{ideal} = FileSize$). The significance of Sync, ChunkOrder, and Fsync imply that certain write patterns are much worse than others for ext4 allocator.

Factor prioritization gives us an overview of the importance of each factor and guides further exploration. We would also like to know which factors and which levels of a factor are most responsible for the tail. This can be determined with *factor mapping* [38]; factor mapping uses a threshold value to group responses (i.e., *d-span* values) into tail and non-tail categories and finds the input space of factors that drive the system into each category. We define the threshold value as the 90th% (10GB in this case)



(a) DiskSize **(b)** FileSize **(c)** Sync **(d)** ChunkOrder

(e) Fsync **(f)** Freespacelayout **(g)** UsedRatio **(h)** DirectorySpan **(i)** InternalDensity **(j)** FileCount **(k)** CPUCount

Figure 5: Tail Distribution of 11 Factors. In the figure, we can find what levels of each factor have tail runs and percentage of tail runs in each level. Regions with significantly more tail runs are marked bold. Note that the number of total runs of each level is identical for each factor. Therefore, the percentages between levels of a factor are comparable. For example, (a) shows all tail runs in the experiment have disk sizes $\geq$ 16GB. In addition, when DiskSize=16GB, 17% of runs are in the tail (*d-span*$\geq$10GB) which is less than DiskSize=32GB.

of all *d-span*s in the experiment. We say that a run is a *tail run* if its response is in the tail category.

Factor mapping visualization in Figure 5 shows how the tails are distributed to the levels of each factor. Thanks to the balanced Latin hypercube design with large sample size, the difference between any two levels of a factor is likely to be attributed to the level change of this factor and not due to chance.

Figure 5a shows that all tail runs lay on disk sizes over 8GB because the threshold *d-span* (10GB) is only possible when the disk size exceeds that size. This result implies that blocks are spread farther as the capacity of the disk increases, possibly due to poor allocation polices in ext4. Figure 5b shows a surprising result: there are significantly more tail runs when the file size is larger than 64KB. This reveals that ext4 uses very different block allocation polices for files below and above 64KB.

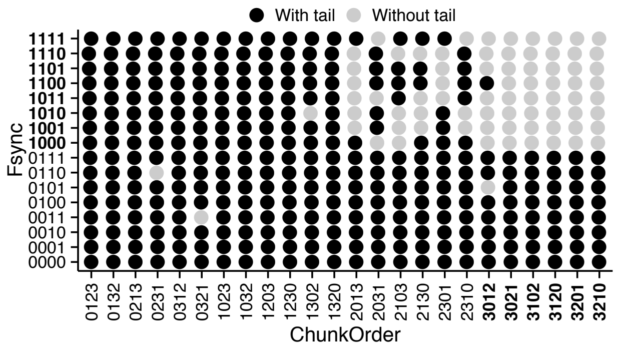
Sync, ChunkOrder, and Fsync also present interesting behaviors, in which the first written chunk plays an important role in deciding the tail. Figure 5c shows that closing and sync-ing after the first written chunk (coded 1***) causes more tail runs than otherwise. Figure 5d shows that writing chunk-0 of a file first (coded 0***), including sequential writes (coded 0123) which are usually preferred, leads to more tail runs. Figure 5e shows that, on average, not fsync-ing the first written chunk (coded 0***) leads to more tail runs than otherwise.

The rest of the factors are less significant, but still reveal interesting observations. Figure 5f and Figure 5g show that tail runs are always present and not strongly correlated with free space layout or the amount of free space, even given the small file sizes in our workloads (below 256KB). Even with layout number 6 (not manually fragmented), there are still many tail runs. Similarly, having more free spaces does not reduce tail cases. These facts indicate that many tail runs do not depend on the disk state and instead it is the ext4 block allocation policy itself causing these tail runs. After we fix the ext4 allocation polices in the next section, the DiskUsed and FreespaceLayout factors will have a much stronger impact.

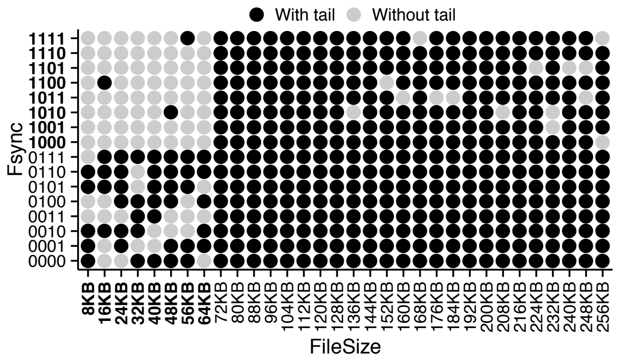
Finally, Figure 5h and Figure 5i show that tail runs are generally not affected by DirectorySpan and InternalDensity. Figure 5j shows that having more files leads to 29% more tail cases, indicating potential layout problems in production systems where multi-file operations are common. Figure 5k shows that there are 6% more tail cases when there are two CPUs.

3.1.3 Which factors interact in the tail?

In a complex system such as ext4 block allocator, performance may depend on more than one factor. We have inspected all two-factor interactions and select two cases in Figure 6 that present clear patterns. The figures show how pairwise interactions may lead to tail runs, reveal-



(a) ChunkOrder and Fsync.



(b) FileSize and Fsync.

Figure 6: Tail Runs in the Interactions of Factors Note that each interaction data point corresponds to multiple runs with other factors varying. A black dot means that there is at least one tail case in that interaction. Low-danger zones are marked with bold labels.

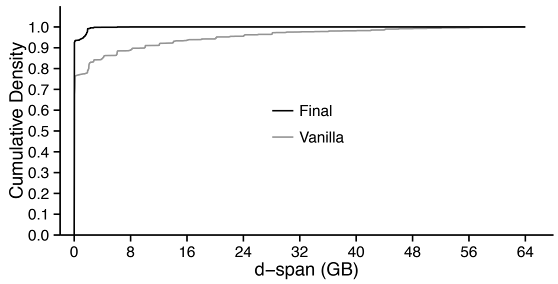
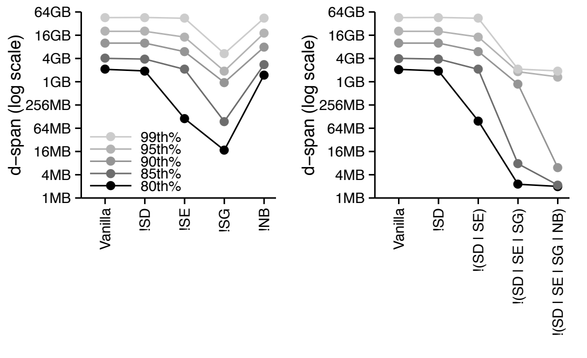


Figure 7: *d-span* CDF of vanilla and final versions of ext4. The final version reduces the 80th, 90th, and 99th percentiles by 1.0×10^3, 1.6×10^3, and 24 times, respectively.



(a) Effect of Single Fix **(b)** Cumulative Effect of Fixes

Figure 8: Effect of fixing issues. Vanilla: Linux v3.12.5. "!" means "without". *SD*: Scheduler Dependency; *SE*: Special End; *SG*: Shared Goal; *NB*: Normalization Bug. !(X | Y) means X and Y are both removed in this version.

ing both dangerous and low-danger zones in the workload space; these zones give us hints about the causes of the tail, which will be investigated in Section 3.2. Figure 6a shows that, writing and fsync-ing chunk-3 first significantly reduces tail cases. In Figure 6b, we see that, for files not larger than 64KB, fsync-ing the first written chunk significantly reduces the possibility of producing tail runs. These two figures do not conflict with each other; in fact, they indicate a low-danger zone in a three-dimension space.

Evaluating ext4 as black box, we have shown that ext4 does not consistently provide good layouts given diverse inputs. Our results show that unstable performance with ext4 is not due to the external state of the disk (e.g., fragmentation or utilization), but to the internal policies of ext4. To understand and fix the problems with ext4 allocation, we use detailed results from *Chopper* to guide our search through ext4 documentation and source code.

3.2 File System as a White Box

Our previous analysis uncovered a number of problems with the layout policies of ext4, but it did not pinpoint the location of those policies within the ext4 source code. We now use the hints provided by our previous data analysis to narrow down the sources of problems and to perform detailed source code tracing given the set of workloads suggested by *Chopper*. In this manner, we are able to fix a series of problems in the ext4 layout policies and show that each fix reduces the tail cases in ext4 layout.

Figure 7 compares the original version of ext4 and our final version that has four sources of layout variation removed. We can see that the fixes significantly reduce the size of the tail, providing better and more consistent layout quality. We now connect the symptoms of problems shown by *Chopper* to their root causes in the code.

3.2.1 Randomness $\rightarrow$ Scheduler Dependency

Our first step is to remove non-determinism for experiments with the same treatment. Our previous experiments corresponded to a single run for each treatment; this approach was acceptable for summarizing from a large sample space, but cannot show intra-treatment variation. After we identify and remove this intra-treatment variation, it will be more straightforward to remove other tail effects.

We conducted two repeated experiments with the same input space as in Table 1 and found that 6% of the runs have different *d-span*s for the same treatment; thus, ext4 can produce different layouts for the same controlled input. Figure 9a shows the distribution of the *d-span* differences for those 6% of runs. The graph indicates that the physical data layout can differ by as much as 46GB for the same workload.

(a)

(b)

(c)

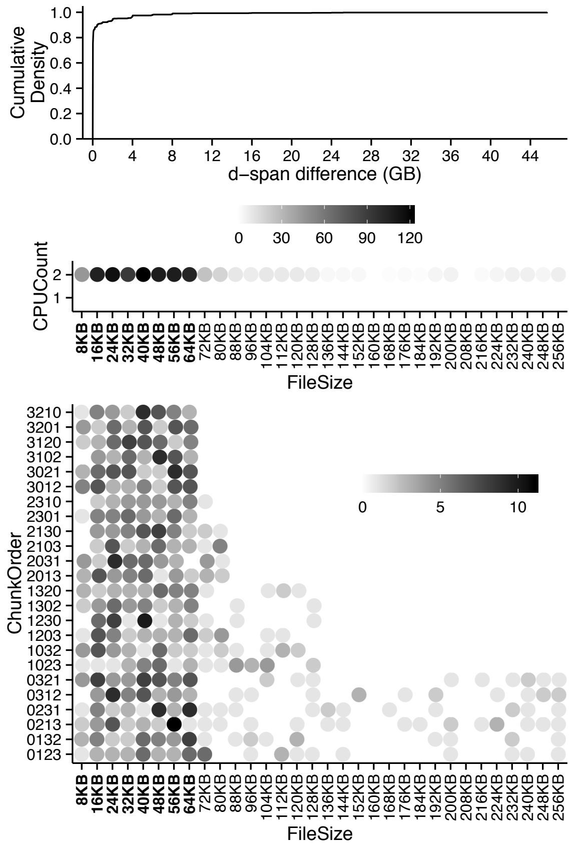


Figure 9: Symptoms of Randomness. (a): CDF of *d-span* variations between two experiments. The median is 1.9MB. The max is 46GB. (b): Number of runs with changed *d-span*, shown as the interaction of FileSize and CPUCount. (c): Number of runs with changed *d-span*, shown as the interaction of FileSize and ChunkOrder. Regions with considerable tail runs are marked with bold labels.

Examining the full set of factors responsible for this variation, we found interesting interactions between FileSize, CPUCount, and ChunkOrder. Figure 9b shows the count of runs in which *d-span* changed between identical treatments as a function of CPUCount and FileSize. This figure gives us the hint that *small* files in multiple-CPU systems may suffer from unpredictable layouts. Figure 9c shows the number of runs with changed *d-span* as a function of ChunkOrder and FileSize. This figure indicates that most small files and those large files written with more sequential patterns are affected.

Root Cause: With these symptoms as hints we focused on the interaction between small files and the CPU scheduler. Linux ext4 has an allocation policy such that files not larger than 64KB (*small* files) are allocated from *locality group (LG) preallocations*; further, the block allocator associates each LG preallocation with a CPU, in order to avoid contention. Thus, for *small* files, the layout location is based solely on which CPU the flusher thread is running. Since the flusher thread can be scheduled on different CPUs, the same small file can use different LG preallocations spread across the entire disk.

This policy is also the cause of the variation seen by some large files written sequentially: large files written sequentially begin as small files and are subject to LG preallocation; large files written backwards have large sizes from the beginning and never trigger this scheduling dependency[1]. In production systems with heavy loads, more cores, and more files, we expect more unexpected poor layouts due to this effect.

Fix: We remove the problem of random layout by choosing the locality group for a *small* file based on its i-number range instead of the CPU. Using the i-number not only removes the dependency on the scheduler, but also ensures that *small* files with close i-numbers are likely to be placed close together. We refer to the ext4 version with this new policy as *!SD*, for no Scheduler Dependency.

Figure 8a compares vanilla ext4 and *!SD*. The graph shows that the new version slightly reduces the size of the tail. Further analysis shows that in total *d-span* is reduced by 1.4 TB in 7% of the runs but is increased by 0.8 TB in 3% of runs. These mixed results occur because this first fix unmasks other problems which can lead to larger *d-span*s. In complex systems such as ext4, performance problems interact in surprising ways; we will progressively work to remove three more problems.

3.2.2 Allocating Last Chunk $\rightarrow$ Special End

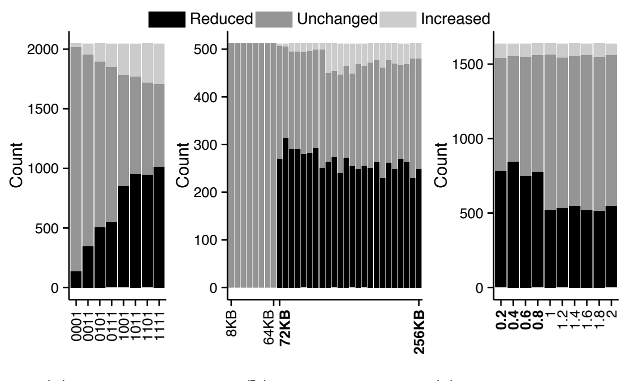
We now return to the interesting behaviors originally shown in Figure 6a, which showed that allocating chunk-3 first (Fsync=1*** and ChunkOrder=3***) helps to avoid tail runs. To determine the cause of poor allocations, we compared traces from selected workloads in which a tail occurs to similar workloads in which tails do not occur.

Root Cause: Linux ext4 uses a *Special End* policy to allocate the last extent of a file when the file is no longer open; specifically, the last extent does not trigger preallocation. The Special End policy is implemented by checking three conditions - *Condition 1*: the extent is at the end of the file; *Condition 2*: the file system is not busy; *Condition 3*: the file is not open. If all conditions are satisfied, this request is marked with the hint "do not preallocate", which is different from other parts of the file[2].

The motivation is that, since the status of a file is final (i.e., no process can change the file until the next open), there is no need to reserve additional space. While this motivation is valid, the implementation causes an inconsistent allocation for the last extent of the file compared to the rest; the consequence is that blocks can be spread

[1]Note that file size in ext4 is calculated by the ending logical block number of the file, not the sum of physical blocks occupied.

[2]In fact, this hint is vague. It means: 1. if there is a preallocation solely for this file (i.e., *i-node preallocation*), use it; 2. do not use LG preallocations, even they are available 3. do not create any new preallocations.



(a) Sync **(b)** FileSize **(c)** InternalDensity

Figure 10: Effects of removing problematic policies. The *d-span*s could be 'Reduced', 'Unchanged' or 'Increased' due to the removal. (a): removing Special End; (b) and (c): removing Shared Global.

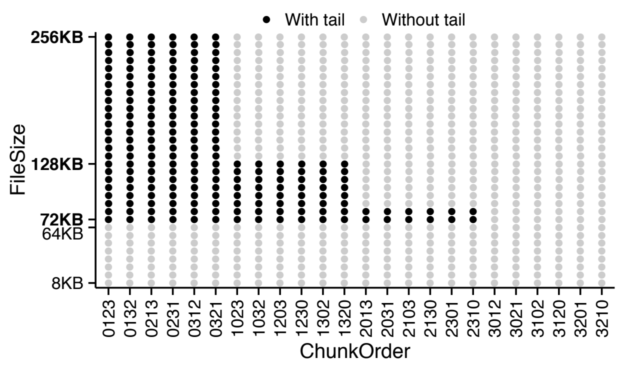


Figure 11: Tail Runs in *!(SD|SE)*. The figure shows tail runs in the interaction of ChunkOrder and FileSize, after removing Scheduler Dependency and Special End.

far apart. For example, a small file may be inadvertently split because non-ending extents are allocated with LG preallocations while the ending extent is not; thus, these conflicting policies drag the extents of the file apart.

This policy explains the tail-free zone (Fsync=1*** and ChunkOrder=3***) in Figure 6a. In these tail-free zones, the three conditions cannot be simultaneously satisfied since fsync-ing chunk-3 causes the last extent to be allocated, while the file is still open; thus, the Special End policy is not triggered.

Fix: To reduce the layout variability, we have removed the Special End policy from ext4; in this version named *!SE*, the ending extent is treated like all other parts of the file. Figure 8 shows that *!SE* reduces the size of the tail. Further analysis of the results show that removing Special End policy reduces *d-span*s for 32% of the runs by a total of 21TB, but increases *d-span*s for 14% of the runs by a total of 9TB. The increasing of *d-span* is primarily because removing this policy unmasks inconsistent policies in File Size Dependency, which we will discuss next.

Figure 10a examines the benefits of the *!SE* policy compared to vanilla ext4 in more detail; to compare only deterministic results, we set CPUCount=1. The graph shows that the *!SE* policy significantly reduces tail runs when the workload begins with sync operations (combination of `close()`, `sync()`, and `open()`); this is because the Special End policy is more likely to be triggered when the file is temporarily closed.

3.2.3 File Size Dependency $\rightarrow$ Shared Global

After removing the Scheduler Dependency and Special End policies, ext4 layout still presents a significant tail. Experimenting with these two fixes, we observe a new symptom that occurs due to the interaction of FileSize and ChunkOrder, as shown in Figure 11. The stair shape of the tail runs across workloads indicates that this policy only affects *large* files and it depends upon the first written chunk.

Root Cause: Traces of several representative data points reveal the source of the 'stair' symptom, which we call *File Size Dependency*. In ext4, one of the design goals is to place small files (less than 64KB, which is tunable) close and big files apart [7]. Blocks for *small* files are allocated from *LG preallocations*, which are shared by all *small* files; blocks in *large* files are allocated from per-file *inode preallocations* (except for the ending extent of a closed file, due to the Special End policy).

This file-size-dependant policy ignores the activeness of files, since the dynamically changing size of a file may trigger inconsistent allocation policies for the same file. In other words, blocks of a file larger than 64KB can be allocated with two distinct policies as the file grows from *small* to *large*. This changing policy explains why FileSize is the most significant workload factor, as seen in Figure 4, and why Figure 5b shows such a dramatic change at 64KB.

Sequential writes are likely to trigger this problem. For example, the first 36KB extent of a 72KB file will be allocated from the LG preallocation; the next 36KB extent will be allocated from a new i-node preallocation (since the file is now classified as *large* with 72KB $>$ 64KB). The allocator will try to allocate the second extent next to the first, but the preferred location is already occupied by the LG preallocation; the next choice is to use the block group where the last big file in the whole file system was allocated (Shared Global policy, coded *SG*), which can be far away. Growing a file often triggers this problem. File Size Dependency is the reason why runs with ChunkOrder=0*** in Figure 5d and Figure 11 have relatively more tail runs than other orders. Writing Chunk-0 first makes the file grow from a *small* size and increases the chance of triggering two distinct policies.

Fix: Placing extents of *large* files together with a shared global policy violates the initial design goal of placing big files apart and deteriorates the consequences of File Size Dependency. To mitigate the problem, we implemented a new policy (coded *!SG*) that tries to place

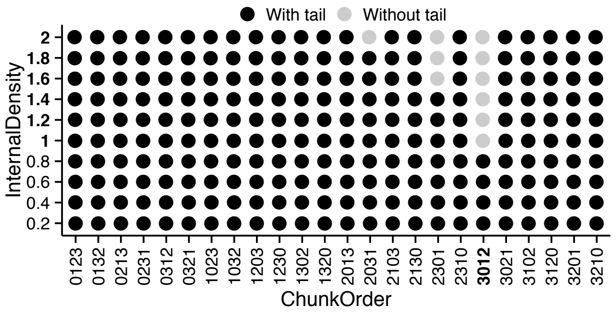


Figure 12: Tail Runs in *!(SD|SE|SG)*. This figure shows tail runs in the interaction of ChunkOrder and InternalDensity on version *!(SD|SE|SG)*.

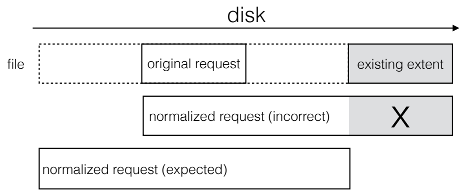


Figure 13: Ill Implementation of Request Normalization. In this case, the normalized request overlaps with the existing extent of the file, making it impossible to fulfill the request at the preferred location.

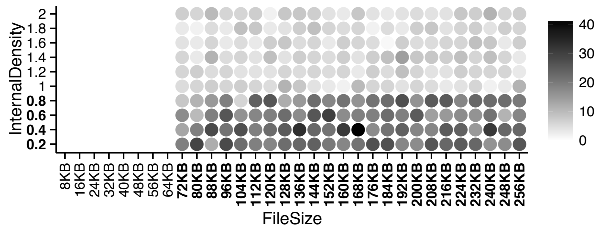


Figure 14: Impact of Normalization Bug. This figure shows the count of runs affected by Normalization Bug in the interaction of FileSize and InternalDensity. The count is obtained by comparing experimental results ran with and without the bug.

extents of *large* files close to existing extents of that file. Figure 8a shows that *!SG* significantly reduces the size of the tail. In more detail, *!SG* reduces *d-span* in 35% of the runs by a total of 45TB.

To demonstrate the effectiveness of the *!SG* version, we compare the number of tail cases with it and vanilla ext4 for deterministic scenarios (CPUCount=1). Figure 10b shows that the layout of *large* files (>64KB) is significantly improved with this fix. Figure 10c shows that the layout of sparse files (with InternalDensity < 1) is also improved; the new policy is able to separately allocate each extent while still keeping them near one another.

3.2.4 Sparse Files → Normalization Bug

With three problems fixed in version *!(SD|SE|SG)*, we show an interesting interaction that still remains between ChunkOrder and InternalDensity. Figure 12 shows that while most of the workloads exhibit tails, several workloads do not, specifically, all "solid" (InternalDensity≥1) files with ChunkOrder=3012. To identify the root cause, we focus only on workloads with ChunkOrder=3012 and compare solid and sparse patterns.

Root Cause: Comparing solid and sparse runs with ChunkOrder=3012 shows that the source of the tail is a bug in ext4 normalization; normalization enlarges requests so that the extra space can be used for a similar extent later. The normalization function should update the request's logical starting block number, corresponding physical block number, and size; however, with the bug, the physical block number is not updated and the old value is used later for allocation[3].

Figure 13 illustrates how this bug can lead to poor layout. In this scenario, an ill-normalized request is started (incorrectly) at the original physical block number, but is of a new (correct) larger size; as a result, the request will not fit in the desired gap within this file. Therefore, ext4 may fail to allocate blocks from preferred locations and will perform a desperate search for free space elsewhere, spreading blocks. The solid files with ChunkOrder of 3012 in Figure 12 avoid this bug because if chunks-0,1,2 are written sequentially after chunk-3 exists, then the physical block number of the request does not need to be updated.

Fix: We fix the bug by correctly updating the physical starting block of the request in version *!NB*. Figure 14 shows that *large* files were particularly susceptible to this bug, as were sparse files ($InternalDensity < 1$). Figure 8a shows that fixing this bug reduces the tail cases, as desired. In more detail, *!NB* reduces *d-span* for 19% of runs by 8.3 TB in total. Surprisingly, fixing the bug increases *d-span* for 5% of runs by 1.5 TB in total. Trace analysis reveals that, by pure luck, the mis-implemented normalization sometimes sets the request to nearby space which happened to be free, while the correct request fell in space occupied by another file; thus, with the correct request, ext4 sometimes performs a desperate search and chooses a more distant location.

Figure 8 summarizes the benefits of these four fixes. Overall, with all four fixes, the 90th-percentile for *d-span* values is dramatically reduced from well over 4GB to close to 4MB. Thus, as originally shown in Figure 7, our final version of ext4 has a much less significant tail than the original ext4.

3.3 Latencies Reduced

Chopper uses *d-span* as a diagnostic signal to find problematic block allocator designs that produce poor data layouts. The poor layouts, which incur costly disk seeks on HDDs [36], garbage collections on SSDs [11] and even

[3]This bug is present even in the currently latest version of Linux, Linux v3.17-rc6. It has been confirmed by an ext4 developer and is waiting for further tests.

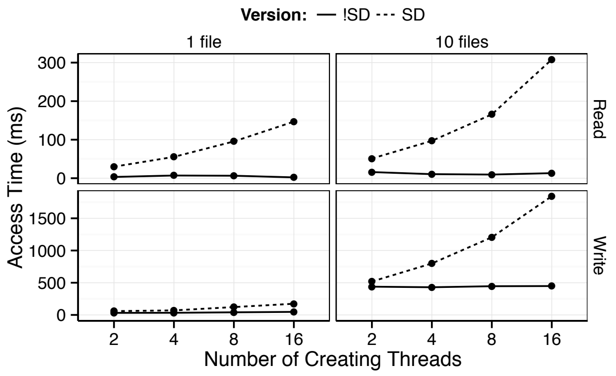


Figure 15: Latency Reduction. This figure shows that *!SD* significantly reduces average data access time comparing with *SD*. All experiments were repeated 5 times. Standard errors are small and thus hidden for clarity.

CPU spikes [1], can in turn result in long data access latencies. Our repairs based on *Chopper*'s findings reduce latencies caused by the problematic designs.

For example, Figure 15 demonstrates how Scheduler Dependency incurs long latencies and how our repaired version, *!SD*, reduces latencies on an HDD (Hitachi HUA723030ALA640: 3.0 TB, 7200 RPM). In the experiment, files were created by multiple *creating threads* residing on different CPUs; each of the threads wrote a part of a 64KB file. We then measured file access time by reading and over-writing with one thread, which avoids resource contentions and maximizes performance. To obtain application-disk data transfer performance, OS and disk cache effects were circumvented. Figure 15 shows that with the *SD* version, access time increases with more creating threads because SD splits each file into more and potentially distant physical data pieces. Our fixed version, *!SD*, reduced read and write time by up to 67 and 4 times proportionally, and by up to 300 and 1400 ms. The reductions in this experiment, as well as expected greater ones with more creating threads and files, are significant – as a comparison, a round trip between US and Europe for a network packet takes 150 ms and a round trip within the same data center takes 0.5 ms [22, 32]. The time increase caused by Scheduler Dependency, as well as other issues, may translate to long latencies in high-level data center operations [17]. *Chopper* is able to find such issues, leading to fixes reducing latencies.

3.4 Discussion

With the help of exploratory data analysis, we have found and removed four issues in ext4 that can lead to unexpected tail latencies; these issues are summarized in Table 2. We have made the patches for these issues publicly available with *Chopper*.

While these fixes do significantly reduce the tail behaviors, they have several potential limitations. First, without the Scheduler Dependency policy, flusher threads running on different CPUs may contend for the same preallocation groups. We believe that the contention degree is acceptable, since allocation within a preallocation is fast and files are distributed across many preallocations; if contention is found to be a problem, more preallocations can be added (the current ext4 creates preallocations lazily, one for each CPU). Second, removing the Shared Global policy mitigates but does not eliminate the layout problem for files with dynamically changing sizes; choosing policies based on dynamic properties such as file size is complicated and requires more fundamental policy revisions. Third, our final version, as shown in Figure 7, still contains a small tail. This tail is due to the disk state (DiskUsed and FreespaceLayout); as expected, when the file system is run on a disk that is more heavily used and is more fragmented, the layout for new files suffers.

Issue	Description
Scheduler Dependency	Choice of preallocation group for small files depends on CPU of flushing thread.
Special End	The last extent of a closed file may be rejected to allocate from preallocated spaces.
File Size Dependency	Preferred target locations depend on file size which may dynamically change.
Normalization Bug	Block allocation requests for large files are not correctly adjusted, causing the allocator to examine mis-aligned locations for free space.

Table 2: Linux ext4 Issues. This table summarizes issues we have found and fixed.

The symptoms of internal design problems revealed by *Chopper* drive us to reason about their causes. In this process, time-consuming tracing is often necessary to pinpoint a particular problematic code line as the code makes complex decisions based on environmental factors. Fortunately, analyzing and visualizing the data sets produced by *Chopper* enabled us to focus on several representative runs. In addition, we can easily reproduce and trace any runs in the controlled environmental provided by *Chopper*, without worrying about confounding noises.

With *Chopper*, we have learned several lessons from our experience with ext4 that may help build file systems that are robust to uncertain workload and environmental factors in the future. First, policies for different circumstances should be harmonious with one another. For example, ext4 tries to optimize allocation for different scenarios and as a result has a different policy for each case (e.g., the ending extent, *small* and *large* files); when multiple policies are triggered for the same file, the policies conflict and the file is dragged apart. Second, policies should not depend on environmental factors that may change and are outside the control of the file system. In contrast, data layout in ext4 depends on the OS scheduler, which makes layout quality unpredictable. By simplifying the layout policies in ext4 to avoid special cases and to be independent of environmental factors, we have shown that file layout is much more compact and predictable.

4 Related Work

Chopper is a comprehensive diagnostic tool that provides techniques to explore file system block allocation designs. It shares similarities and has notable differences with traditional benchmarks and with model checkers.

File system benchmarks have been criticized for decades [44–46]. Many file system benchmarks target many aspects of file system performance and thus include many factors that affect the results in unpredictable ways. In contrast, *Chopper* leverages well-developed statistical techniques [37,38,48] to isolate the impact of various factors and avoid noise. With its sole focus on block allocation, *Chopper* is able to isolate its behavior and reveal problems with data layout quality.

The self-scaling I/O benchmark [14] is similar to *Chopper*, but the self-scaling benchmark searches a five-dimension workload parameter space by dynamically adjusting *one parameter* at a time while keeping the rest constant; its goal is to converge all parameters to values that uniformly achieve a specific percentage of max performance, which is called a *focal point*. This approach was able to find interesting behaviors, but it is limited and has several problems. First, the experiments may never find such a focal point. Second, the approach is not feasible given a large number of parameters. Third, changing one parameter at a time may miss interesting points in the space and interactions between parameters. In contrast, *Chopper* has been designed to systematically extract the maximum amount of information from limited runs.

Model checking is a verification process that explores system state space [16]; it has also been used to diagnose latent performance bugs. For example, MacePC [27] can identify bad performance and pinpoint the causing state. One problem with this approach is that it requires a simulation which may not perfectly match the desired implementation. Implementation-level model checkers, such as FiSC [50], address this problem by checking the actual system. FiSC checks a real Linux kernel in a customized environment to find file system bugs; however, FiSC needs to run the whole OS in the model checker and intercept calls. In contrast, *Chopper* can run in an unmodified, low-overhead environment. In addition, *Chopper* explores the input space differently; model checkers consider transitions between states and often use tree search algorithms, which may have clustered exploration states and leave gaps unexplored. In *Chopper*, we precisely define a large number of factors and ensure the effects and interactions of these factors are evenly explored by statistical experimental design [29,37,38,48].

5 Conclusions

Tail behaviors have high consequences and cause unexpected system fluctuations. Removing tail behaviors will lead to a system with more consistent performance. However, identifying tails and finding their sources are challenging in complex systems because the input space can be infinite and exhaustive search is impossible. To study the tails of block allocation in XFS and ext4, we built *Chopper* to facilitate carefully designed experiments to effectively explore the input space of more than ten factors. We used Latin hypercube design and sensitivity analysis to uncover unexpected behaviors among many of those factors. Analysis with *Chopper* helped us pinpoint and remove four layout issues in ext4; our improvements significantly reduce the problematic behaviors causing tail latencies. We have made *Chopper* and ext4 patches publicly available.

We believe that the application of established statistical methodologies to system analysis can have a tremendous impact on system design and implementation. We encourage developers and researchers alike to make systems amenable to such experimentation, as experiments are essential in the analysis and construction of robust systems. Rigorous statistics will help to reduce unexpected issues caused by intuitive but unreliable design decisions.

Acknowledgments

We thank the anonymous reviewers and Angela Demke Brown (our shepherd) for their excellent feedback. This research was supported by the United States Department of Defense, NSF grants CCF-1016924, CNS-1421033, CNS-1319405, CNS-1218405, CNS-1042537, and CNS-1042543 (PRObE http://www.nmc-probe.org/), as well as generous donations from Amazon, Cisco, EMC, Facebook, Fusion-io, Google, Huawei, IBM, Los Alamos National Laboratory, Microsoft, NetApp, Samsung, Sony, Symantec, and VMware. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DOD, NSF, or other institutions.

References

[1] Btrfs Issues. `https://btrfs.wiki.kernel.org/index.php/Gotchas`.

[2] NASA Archival Storage System. `http://www.nas.nasa.gov/hecc/resources/storage_systems.html`.

[3] Red Hat Enterprise Linux 7 Press Release. `http://www.redhat.com/en/about/press-releases/red-hat-unveils-rhel-7`.

[4] Ubuntu. `http://www.ubuntu.com`.

[5] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. A Five-Year Study of File-System Metadata. In *Proceedings of the*

5th USENIX Symposium on File and Storage Technologies (FAST '07), San Jose, California, February 2007.

[6] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 253–266, San Jose, CA, 2012. USENIX.

[7] KV Aneesh Kumar, Mingming Cao, Jose R Santos, and Andreas Dilger. Ext4 block and inode allocator improvements. In *Proceedings of the Linux Symposium*, volume 1, pages 263–274, 2008.

[8] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.8 edition, 2014.

[9] Mary Baker, John Hartman, Martin Kupfer, Ken Shirriff, and John Ousterhout. Measurements of a Distributed File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 198–212, Pacific Grove, California, October 1991.

[10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Communications of the ACM*, February 2010.

[11] Luc Bouganim, Björn Thór Jónsson, Philippe Bonnet, et al. uFLIP: Understanding flash IO patterns. In *4th Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 1–12, 2009.

[12] Rob Carnell. lhs package manual. `http://cran.r-project.org/web/packages/lhs/lhs.pdf`.

[13] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, pages 181–192, New York, NY, USA, 2009. ACM.

[14] Peter M. Chen and David A. Patterson. A New Approach to I/O Performance Evaluation–Self-Scaling I/O Benchmarks, Predicted I/O Performance. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '93)*, pages 1–12, Santa Clara, California, May 1993.

[15] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 228–243, 2013.

[16] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[17] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[18] Nykamp DQ. Mean path length definition. `http://mathinsight.org/network_mean_path_length_definition`.

[19] Dawson Engler and Madanlal Musuvathi. Static Analysis versus Software Model Checking for Bug Finding. In *5th International Conference Verification, Model Checking and Abstract Interpretation (VMCAI '04)*, Venice, Italy, January 2004.

[20] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 29–43, Bolton Landing, New York, October 2003.

[21] Garth Gibson, Gary Grider, Andree Jacobson, and Wyatt Lloyd. PRObE: A Thousand-Node Experimental Cluster for Computer Systems Research. *USENIX ;login:*, 38(3), June 2013.

[22] Brendan Gregg. *Systems Performance: Enterprise and the Cloud*, page 20. 2013.

[23] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A file is not a file: understanding the I/O behavior of Apple desktop applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 71–83. ACM, 2011.

[24] Jon C. Helton and Freddie J. Davis. Latin hypercube sampling and the propagation of uncertainty in analyses of complex systems. *Reliability Engineering & System Safety*, 81(1):23–69, 2003.

[25] Ronald L. Iman, Jon C. Helton, and James E. Campbell. An approach to sensitivity analysis of computer models. Part I - Introduction, Input, Variable Selection and Preliminary Variable Assessment. *Journal of Quality Technology*, 13:174–183, 1981.

[26] Keren Jin and Ethan L. Miller. The Effectiveness of Deduplication on Virtual Machine Disk Images. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, pages 7:1–7:12, 2009.

[27] Charles Killian, Karthik Nagaraj, Salman Pervez, Ryan Braud, James W. Anderson, and Ranjit Jhala. Finding Latent Performance Bugs in Systems Implementations. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 17–26, New York, NY, USA, 2010. ACM.

[28] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Alex Tomas Andreas Dilge and, and Laurent Vivier. The New Ext4 filesystem: Current Status and Future Plans. In *Ottawa Linux Symposium (OLS '07)*, Ottawa, Canada, July 2007.

[29] Michael D. McKay, Richard J. Beckman, and William J. Conover. Comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 21(2):239–245, 1979.

[30] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. SFS: random write considered harmful in solid state drives. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*. USENIX Association, 2012.

[31] V. N. Nair, D. A. James, W. K. Ehrlich, and J. Zevallos. A statistical assessment of some software testing strategies and application of experimental design techniques. *Statistica Sinica*, 8(1):165–184, 1998.

[32] Peter Norvig. Teach Yourself Programming in Ten Years. `http://norvig.com/21-days.html`.

[33] Ryan Paul. Google upgrading to Ext4. `arstechnica.com/information-technology/2010/01/google-upgrading-to-ext4-hires-former-linux-foundation-cto/`.

[34] Zachary N. J. Peterson. Data Placement for Copy-on-write Using Virtual Contiguity. Master's thesis, U.C. Santa Cruz, 2002.

[35] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A Comparison of File System Workloads. In *Proceedings of the USENIX Annual Technical Conference (USENIX '00)*, pages 41–54, San Diego, California, June 2000.

[36] Chris Ruemmler and John Wilkes. An Introduction to Disk Drive Modeling. *IEEE Computer*, 27(3):17–28, March 1994.

[37] Jerome Sacks, William J Welch, Toby J Mitchell, and Henry P Wynn. Design and analysis of computer experiments. *Statistical science*, pages 409–423, 1989.

[38] Andrea Saltelli, Marco Ratto, Terry Andres, Francesca Campolongo, Jessica Cariboni, Debora Gatelli, Michaela Saisana, and Stefano Tarantola. *Global sensitivity analysis: the primer*. John Wiley & Sons, 2008.

[39] Andrea Saltelli, Stefano Tarantola, Francesca Campolongo, and Marco Ratto. *Sensitivity analysis in practice: a guide to assessing scientific models*. John Wiley & Sons, 2004.

[40] Thomas J Santner, Brian J Williams, and William Notz. *The design and analysis of computer experiments*. Springer, 2003.

[41] SGI. XFS Filesystem Structure. `http://oss.sgi.com/projects/xfs/papers/xfs_filesystem_structure.pdf`.

[42] Keith A. Smith and Margo I. Seltzer. File System Aging – Increasing the Relevance of File System Benchmarks. In *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '97, pages 203–213, New York, NY, USA, 1997. ACM.

[43] IM Soboĺ. Quasi-Monte Carlo methods. *Progress in Nuclear Energy*, 24(1):55–61, 1990.

[44] Diane Tang and Margo Seltzer. Lies, damned lies, and file system benchmarks. *VINO: The 1994 Fall Harvest*, 1994.

[45] Vasily Tarasov, Saumitra Bhanage, Erez Zadok, and Margo Seltzer. Benchmarking File System Benchmarking: It *IS* Rocket Science. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 9–9, Berkeley, CA, USA, 2011. USENIX Association.

[46] Avishay Traeger, Erez Zadok, Nikolai Joukov, and Charles P. Wright. A Nine Year Study of File System and Storage Benchmarking. *ACM Transactions on Storage (TOS)*, 4(2), May 2008.

[47] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A

Scalable, High-Performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, Washington, November 2006.

[48] CF Jeff Wu and Michael S Hamada. *Experiments: planning, analysis, and optimization*, volume 552. John Wiley & Sons, 2011.

[49] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding Long Tails in the Cloud. In *NSDI*, pages 329–341, 2013.

[50] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using Model Checking to Find Serious File System Errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, California, December 2004.

[51] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. *SIGCOMM Comput. Commun. Rev.*, 42(4):139–150, August 2012.

Skylight—A Window on Shingled Disk Operation

Abutalib Aghayev   Peter Desnoyers
Northeastern University

Abstract

We introduce Skylight, a novel methodology that combines software and hardware techniques to reverse engineer key properties of drive-managed Shingled Magnetic Recording (SMR) drives. The software part of Skylight measures the latency of controlled I/O operations to infer important properties of drive-managed SMR, including type, structure, and size of the persistent cache; type of cleaning algorithm; type of block mapping; and size of bands. The hardware part of Skylight tracks drive head movements during these tests, using a high-speed camera through an observation window drilled through the cover of the drive. These observations not only confirm inferences from measurements, but resolve ambiguities that arise from the use of latency measurements alone. We show the generality and efficacy of our techniques by running them on top of three emulated and two real SMR drives, discovering valuable performance-relevant details of the behavior of the real SMR drives.

1 Introduction

In the nearly 60 years since the hard disk drive (HDD) has been introduced, it has become the mainstay of computer storage systems. In 2013 the hard drive industry shipped over 400 exabytes [1] of storage, or almost 60 gigabytes for every person on earth. Although facing strong competition from NAND flash-based solid-state drives (SSDs), magnetic disks hold a 10× advantage over flash in both total bits shipped [2] and per-bit cost [3], an advantage that will persist if density improvements continue at current rates.

The most recent growth in disk capacity is the result of improvements to perpendicular magnetic recording (PMR) [4], which has yielded terabyte drives by enabling bits as short as 20 nm in tracks 70 nm wide [5], but further increases will require new technologies [6]. Shingled Magnetic Recording (SMR) [7] is the first such technology to reach market: 5 TB drives are available from Seagate [8] and shipments of 8 TB and 10 TB drives have been announced by Seagate [9] and HGST [10]. Other technologies (Heat-Assisted Magnetic Recording [11] and Bit-Patterned Media [12]) remain in the research stage, and may in fact use shingled recording when they are released [13].

Shingled recording spaces tracks more closely, so they overlap like rows of shingles on a roof, squeezing more tracks and bits onto each platter [7]. The increase in density comes at a cost in complexity, as modifying a disk sector will corrupt other data on the overlapped tracks, requiring copying to avoid data loss [14–17]. Rather than push this work onto the host file system [18,19], SMR drives shipped to date preserve compatibility with existing drives by implementing a Shingle Translation Layer (STL) [20,21] that hides this complexity.

Like an SSD, an SMR drive combines out-of-place writes with dynamic mapping in order to efficiently update data, resulting in a drive with performance much different from that of a Conventional Magnetic Recording (CMR) drive due to seek overhead for out-of-order operations. However unlike SSDs, which have been extensively measured and characterized [22,23], little is known about the behavior and performance of SMR drives and their translation layers, or how to optimize file systems, storage arrays, and applications to best use them.

We introduce a methodology for measuring and characterizing such drives, developing a specific series of micro-benchmarks for this characterization process, much as has been done in the past for conventional drives [24–26]. We augment these timing measurements with a novel technique that tracks actual head movements via high-speed camera and image processing and provides a source of reliable information in cases where timing results are ambiguous.

We validate this methodology on three different emulated drives that use STLs previously described in the literature [20, 21, 27], implemented as a Linux *device mapper target* [28] over a conventional drive, demonstrating accurate inference of properties. We then apply this methodology to 5 TB and 8 TB SMR drives provided by Seagate, inferring the STL algorithm and its properties and providing the first public characterization of such drives.

Using our approach we are able to discover important characteristics of the Seagate SMR drives and their translation layer, including the following:

Cache type and size: The drives use a persistent disk cache of 20 GiB and 25 GiB on the 5 TB and 8 TB drives, respectively, with high random write speed until the cache is full. The effective cache size is a function of write size and queue depth.

Persistent cache structure: The persistent disk cache is written as journal entries with quantized sizes—a phenomenon absent from the academic literature on SMRs.

Block Mapping: Non-cached data is statically mapped, using a fixed assignment of logical block addresses (LBAs) to physical block addresses (PBAs), similar to that used in CMR drives, with implications for performance and durability.

Band size: SMR drives organize data in *bands*—a set of contiguous tracks that are re-written as a unit; the examined drives have a small band size of 15–40 MiB.

Cleaning mechanism: Aggressive cleaning during idle times moves data from the persistent cache to bands; cleaning duration is 0.6–1.6 s per modified band.

Our results show the details that may be discovered using Skylight, most of which impact (negatively or positively) the performance of different workloads, as described in § 6. These results—and the toolset allowing similar measurements on new drives—should thus be useful to users of SMR drives, both in determining what workloads are best suited for these drives and in modifying applications to better use them. In addition, we hope that they will be of use to designers of SMR drives and their translation layers, by illustrating the effects of low-level design decisions on system-level performance.

In the rest of the paper we give an overview of Shingled Magnetic Recording (§ 2) followed by the description of emulated and real drives examined (§ 3). We then present our characterization methodology and apply it to all of the drives (§ 4); finally, we survey related work (§ 5) and present our conclusions (§ 6).

2 Background

Shingled recording is a response to limitations on areal density with perpendicular magnetic recording due to the superparamagnetic limit [6]. In brief, for bits to become smaller, write heads must become narrower, resulting in weaker magnetic fields. This requires lower coercivity (easily recordable) media, which is more vulnerable to bit flips due to thermal noise, requiring larger bits for reliability. As the head gets smaller this minimum bit size gets larger, until it reaches the width of the head and further scaling is impossible.

Several technologies have been proposed to go beyond this limit, of which SMR is the simplest [7]. To decrease the bit size further, SMR reduces the track width while keeping the head size constant, resulting in a head that writes a path several tracks wide. Tracks are then overlapped like rows of shingles on a roof, as seen in Figure 1. Writing these overlapping tracks requires only incremental changes in manufacturing, but much greater system changes, as it becomes impossible to re-write a single sector without destroying data on the overlapped sectors.

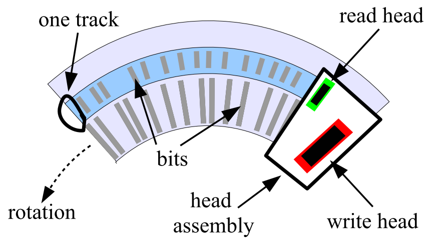


Figure 1: Shingled disk tracks with head width $k = 2$

For maximum capacity an SMR drive could be written from beginning to end, utilizing all tracks. Modifying any of this data, however, would require reading and re-writing the data that would be damaged by that write, and data to be damaged by the re-write, etc. until the end of the surface is reached. This cascade of copying may be halted by inserting *guard regions*—tracks written at the full head width—so that the tracks before the guard region may be re-written without affecting any tracks following it, as shown in Figure 2. These guard regions divide each disk surface into re-writable bands; since the guards hold a single track's worth of data, storage efficiency for a band size of b tracks is $\frac{b}{b+k-1}$.

Given knowledge of these bands, a host file system can ensure they are only written sequentially, for example, by implementing a log-structured file system [18,29]. Standards are being developed to allow a drive to identify these bands to the host [19]: *host-aware* drives report sequential-write-preferred bands (an internal STL handles non-sequential writes), and *host-managed* drives report sequential-write-required bands. These standards are still in draft form, and to date no drives based on them are available on the open market.

Alternately the *drive-managed* disks present a standard re-writable block interface that is implemented by an internal Shingle Translation Layer, much as an SSD uses a Flash Translation Layer (FTL). Although the two are logically similar, appropriate algorithms differ due to differences in the constraints placed by the underlying media: (a) high seek times for non-sequential access, (b) lack of high-speed reads, (c) use of large (10s to 100s of MB) cleaning units, and (d) lack of wear-out, eliminating the need for wear leveling.

These translation layers typically store all data in bands where it is mapped at a coarse granularity, and devote a small fraction of the disk to a *persistent cache*, as shown in Figure 2, which contains copies of recently-written data. Data that should be retrieved from the persistent cache may be identified by checking a *persistent cache map* (or

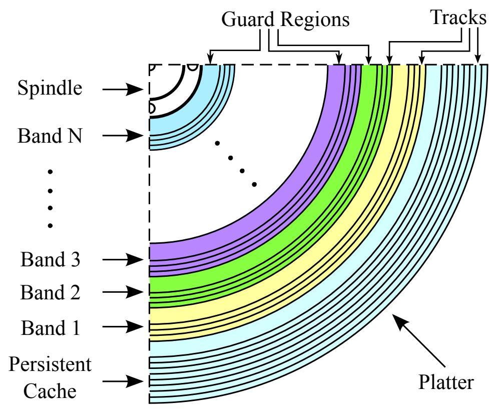


Figure 2: Surface of a platter in a hypothetical SMR drive. A persistent cache consisting of 9 tracks is located at the outer diameter. The guard region that separates the persistent cache from the first band is simply a track that is written at a full head width of k tracks. Although the guard region occupies the width of k tracks, it contains a single track's worth of data and the remaining k-1 tracks are wasted. The bands consist of 4 tracks, also separated with a guard region. Overwriting a sector in the last track of any band will not affect the following band. Overwriting a sector in any of the tracks will require reading and re-writing all of the tracks starting at the affected track and ending at the guard region within the band.

exception map) [20, 21]. Data is moved back from the persistent cache to bands by the process of *cleaning*, which performs read-modify-write (RMW) on every band whose data was overwritten. The cleaning process may be *lazy*, running only when the free cache space is low, or *aggressive*, running during idle times.

In one translation approach, a static mapping algorithmically assigns a *native location* [20] (a PBA) to each LBA in the same way as is done in a CMR drive. An alternate approach uses coarse-grained dynamic mapping for non-cached LBAs [20], in combination with a small number of free bands. During cleaning, the drive writes an updated band to one of these free bands and then updates the dynamic map, potentially eliminating the need for a temporary staging area for cleaning updates and sequential writes.

In any of these cases drive operation may change based on the setting of the *volatile cache* (enabled or disabled) [30]. When the volatile cache is disabled, writes are required to be persistent before completion is reported to the host. When it is enabled, persistence is only guaranteed after a FLUSH command or a write command with the flush (FUA) flag set.

3 Test Drives

We now describe the drives we study. First, we discuss how we emulate three SMR drives using our implementation of two STLs described in the literature. Second, we describe the real SMR drives we study in this paper and the real CMR drive we use for emulating SMR drives.

3.1 Emulated Drives

We implement Cassuto et al.'s *set-associative* STL [20] and a variant of their *S-blocks* STL [20, 31], which we call *fully-associative* STL, as Linux device mapper targets. These are kernel modules that export a pseudo block device to user-space that internally behaves like a drive-managed SMR—the module translates incoming requests using the translation algorithm and executes them on a CMR drive.

The *set-associative* STL manages the disk as a set of N iso-capacity (same-sized) data bands, with typical sizes of 20–40 MiB, and uses a small (1–10%) section of the disk as the persistent cache. The persistent cache is also managed as a set of n iso-capacity cache bands where $n \ll N$. When a block in data band a is to be written, a cache band chosen through $(a \bmod n)$; the next empty block in this cache band is written and the persistent cache map is updated. Further accesses to the block are served from the cache band until cleaning moves the block to its native location, which happens when the cache band becomes full.

The *fully-associative* STL, on the other hand, divides the disk into large (we used 40 GiB) zones and manages each zone independently. A zone starts with 5% of its capacity provisioned to free bands for handling updates. When a block in logical band a is to be written to the corresponding physical band b, a free band c is chosen and written to and the persistent cache map is updated. When the number of free bands falls below a threshold, cleaning merges the bands b and c and writes it to a new band d and remaps the logical band a to the physical band d, freeing bands b and c in the process. This dynamic mapping of bands allows the fully-associative STL to handle streaming writes with zero overhead.

To evaluate the accuracy of our emulation strategy, we implemented a pass-through device mapper target and found negligible overhead for our tests, confirming a previous study [32]. Although in theory, this emulation approach may seem disadvantaged by the lack of access to exact sector layout, in practice this is not the case—even in real SMR drives, the STL running inside the drive is implemented on top of a layer that provides linear PBAs by hiding sector layout and defect management [33]. Therefore, we believe that the device mapper target running on top of a CMR drive provides an accurate model for predicting the behavior of an STL implemented by the controller of an SMR drive.

Table 1 shows the three emulated SMR drive configurations we use in our tests. The first two drives use the set-associative STL, and they differ in the type of persistent cache and band size. The last drive uses the fully-associative STL and disk for the persistent cache. We do not have a drive configuration combining the fully-associative STL and flash for persistent cache, since the fully-associative STL was designed for a drive with a disk cache and uses multiple disk caches evenly spread out on a disk to avoid long seeks during cleaning.

Drive Name	STL	Persistent Cache Type and Size	Disk Cache Multiplicity	Cleaning Type	Band Size	Mapping Type	Size
Emulated-SMR-1	Set-associative	Disk, 37.2 GiB	Single at ID	Lazy	40 MiB	Static	3.9 TB
Emulated-SMR-2	Set-associative	Flash, 9.98 GiB	N/A	Lazy	25 MiB	Static	3.9 TB
Emulated-SMR-3	Fully-associative	Disk, 37.2 GiB	Multiple	Aggressive	20 MiB	Dynamic	3.9 TB

Table 1: Emulated SMR drive configurations.

To emulate an SMR drive with a flash cache (Emulate-SMR-2) we use the Emulate-SMR-1 implementation, but use a device mapper `linear` target to redirect the underlying LBAs corresponding to the persistent cache, storing them on an SSD.

To check the correctness of the emulated SMR drives we ran repeated burn-in tests using `fio` [34]. We also formatted emulated drives with `ext4`, compiled the Linux kernel on top, and successfully booted the system with the compiled kernel. The source code for STLs (1,200 lines of C) and a testing framework (250 lines of Go) are available at `http://sssl.ccs.neu.edu/skylight`.

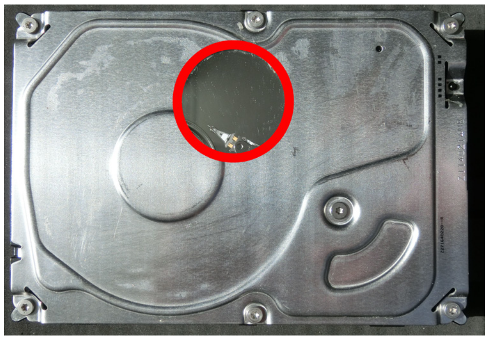


Figure 3: SMR drive with the observation window encircled in red. Head assembly is visible parked at the inner diameter.

3.2 Real Drives

Two real SMR drives were tested: Seagate ST5000AS0011, a 5900 RPM desktop drive (rotation time $\approx$ 10 ms) with four platters, eight heads, and 5 TB capacity (termed Seagate-SMR below), and Seagate ST8000AS0011, a similar drive with six platters, twelve heads and 8 TB capacity. Emulated drives use a Seagate ST4000NC001 (Seagate-CMR), a real CMR drive identical in drive mechanics and specification (except the 4 TB capacity) to the ST5000AS0011. Results for the 8 TB and 5 TB SMR drives were similar; to save space, we only present results for the publicly-available 5 TB drive.

4 Characterization Tests

To motivate our drive characterization methodology we first describe the goals of our measurements. We then describe the mechanisms and methodology for the tests, and finally present results for each tested drive. For emulated SMR drives, we show that the tests produce accurate answers, based on implemented parameters; for real SMR drives we discover their properties. The behavior of the real SMR drives under some of the tests engenders further investigation, leading to the discovery of important details about their operation.

4.1 Characterization Goals

The goal of our measurements is to determine key drive characteristics and parameters:

Drive type: In the absence of information from the vendor, is a drive an SMR or a CMR?

Persistent cache type: Does the drive use flash or disk for the persistent cache? The type of the persistent cache affects the performance of random writes and reliable—volatile cache-disabled—sequential writes. If the drive uses disk for persistent cache, is it a single cache, or is it distributed across the drive [20, 31]? The layout of the persistent disk cache affects the cleaning performance and the performance of the sequential read of a sparsely overwritten linear region.

Cleaning: Does the drive use aggressive cleaning, improving performance for low duty-cycle applications, or lazy cleaning, which may be better for throughput-oriented ones? Can we predict the performance impact of cleaning?

Persistent cache size: After some number of out-of-place writes the drive will need to begin a cleaning process, moving data from the persistent cache to bands so that it can accept new writes, negatively affecting performance. What is this limit, as a function of total blocks written, number of write operations, and other factors?

Band size: Since a band is the smallest unit that may be re-written efficiently, knowledge of band size is important for optimizing SMR drive workloads [20, 27]. What are the band sizes for a drive, and are these sizes constant over time and space [35]?

Block mapping: The mapping type affects performance of both cleaning and reliable sequential writes. For LBAs that are not in the persistent cache, is there a static mapping from LBAs to PBAs, or is this mapping dynamic?

Zone structure: Determining the zone structure of a drive is a common step in understanding block mapping and band size, although the structure itself has little effect on external performance.

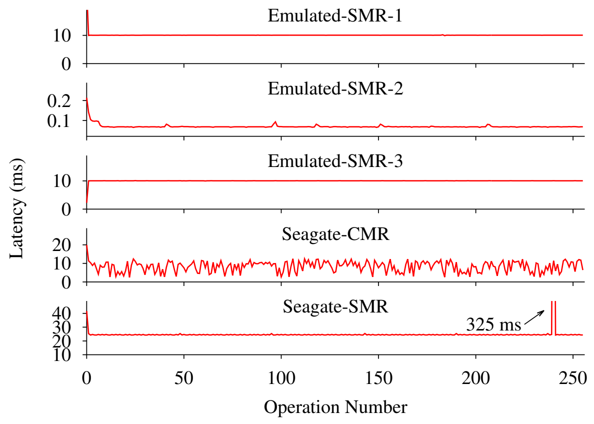


Figure 4: Discovering drive type using latency of random writes. Y-axis varies in each graph.

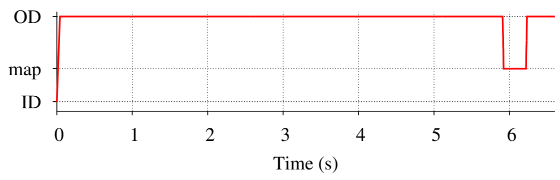


Figure 5: Seagate-SMR head position during random writes.

4.2 Test Mechanisms

The software part of Skylight uses `fio` to generate micro-benchmarks that elicit the drive characteristics. The hardware part of Skylight tracks the head movement during these tests. It resolves ambiguities in the interpretation of the latency data obtained from the micro-benchmarks and leads to discoveries that are not possible with micro-benchmarks alone. To make head tracking possible, we installed (under clean-room conditions) a transparent window in the drive casing over the region traversed by the head. Figure 3 shows the head assembly parked at the inner diameter (ID). We recorded the head movements using Casio EX-ZR500 camera at 1,000 frames per second and processed the recordings with `ffmpeg` to generate head location value for each video frame.

We ran the tests on a 64-bit Intel Core-i3 Haswell system with 16 GiB RAM and 64-bit Linux kernel version 3.14. Unless otherwise stated, we disabled kernel read-ahead, drive look-ahead and drive volatile cache using `hdparm`.

Extensions to `fio` developed for these tests have been integrated back and are available in the latest `fio` release. Slow-motion clips for the head position graphs shown in the paper, as well as the tests themselves, are available at `http://sssl.ccs.neu.edu/skylight`.

4.3 Drive Type and Persistent Cache Type

Test 1 exploits the unusual random write behavior of the SMR drives to differentiate them from CMR drives. While random writes to a CMR drive incur varying latency due to random seek time and rotational delay, random writes to an SMR drive are sequentially logged to the persistent cache with a fixed latency. If random writes are not local, SMR drives using separate persistent caches by the LBA range [20] may still incur varying write latency. Therefore, random writes are done within a small region to ensure that a single persistent cache is used.

Test 1: Discovering Drive Type

```
1 Write blocks in the first 1 GiB in random order to the drive.
2 if latency is fixed then the drive is SMR else the drive is CMR.
```

Figure 4 shows the results for this test. Emulated-SMR-1 sequentially writes incoming random writes to the persistent cache. It fills one empty block after another and due to synchronicity of the writes it misses the next empty block by the time the next write arrives. Therefore, it waits for a complete rotation resulting in a 10 ms write latency, which is the rotation time of the underlying CMR drive. The sub-millisecond latency of Emulated-SMR-2 shows that this drive uses flash for the persistent cache. The latency of Emulated-SMR-3 is identical to that of Emulated-SMR-1, suggesting a similar setup. The varying latency of Seagate-CMR identifies it as a conventional drive. Seagate-SMR shows a fixed ≈ 25 ms latency with a ≈ 325 ms bump at the 240th write. While the fixed latency indicates that it is an SMR drive, we resort to the head position graph to understand why it takes 25 ms to write a single block and what causes the 325 ms latency.

Figure 5 shows that the head, initially parked at the ID, seeks to the outer diameter (OD) for the first write. It stays there during the first 239 writes (incidentally, showing that the persistent cache is at the OD), and on the 240th write it seeks to the center, staying there for ≈ 285 ms before seeking back and continuing to write.

Is all of 25 ms latency associated with every block write spent writing or is some of it spent in rotational delay? When we repeat the test multiple times, the completion time of the first write ranges between 41 and 52 ms, while the remaining writes complete in 25 ms. The latency of the first write always consists of a seek from the ID to the OD (≈ 16 ms). We presume that the remaining time is spent in rotational delay—likely waiting for the beginning of a delimited location—and writing (25 ms). Depending on where the head lands after the seek, the latency of the first write changes between 41 ms and 52 ms. The remaining writes are written as they arrive, without seek time and rotational delay, each taking 25 ms. Hence, a single block *host write* results in a 2.5 track *internal write*. In the following section we explore this phenomenon further.

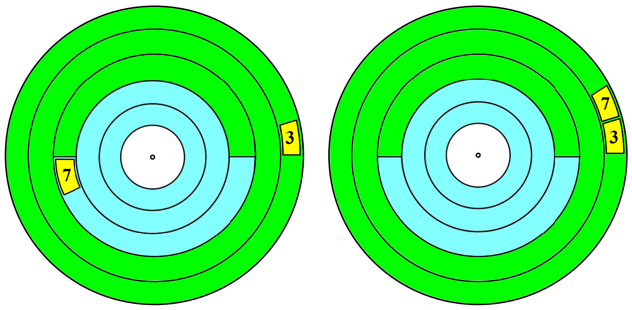


Figure 6: Surface of a disk platter in a hypothetical SMR drive divided into two 2.5 track imaginary regions. The left figure shows the placement of random blocks 3 and 7 when writing synchronously. Each internal write contains a single block and takes 25 ms (50 ms in total) to complete. The drive reports 25 ms write latency for each block; reading the blocks in the written order results in a 5 ms latency. The right figure shows the placement of blocks when writing asynchronously with high queue depth. A single internal write contains both of the blocks, taking 25 ms to complete. The drive still reports 25 ms write latency for each block; reading the blocks back in the written order results in a 10 ms latency due to missed rotation.

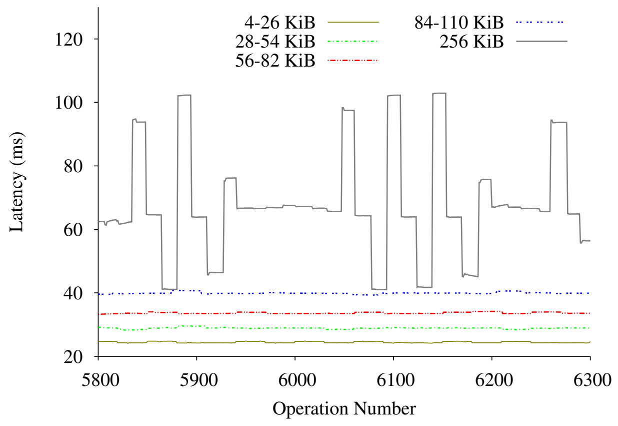


Figure 7: Random write latency of different write sizes on Seagate-SMR, when writing at the queue depth of 31.

4.3.1 Journal Entries with Quantized Sizes

If after Test 1 we immediately read blocks in the written order, read latency is fixed at $\approx$5 ms, indicating 0.5 track distance (covering a complete track takes a full rotation, which is 10 ms for the drive; therefore 5 ms translates to 0.5 track distance) between blocks. On the other hand, if we write blocks asynchronously at the maximum queue depth of 31 [36] and immediately read them, latency is fixed at $\approx$10 ms, indicating a missed rotation due to contiguous placement. Furthermore, although the drive still reports 25 ms completion time for every write, asynchronous writes complete faster—for the 256 write operations, asynchronous writes complete in 216 ms whereas synchronous writes complete in 6,539 ms, as seen in Figure 5. Gathering these facts, we arrive at Figure 6. Writing asynchronously with high queue depth allows the drive to pack multiple blocks into a single internal write, placing them contiguously (shown on the right). The drive reports the completion of individual host writes packed into the same internal write once the internal write completes. Thus, although each of the host writes in the same internal write is reported to take 25 ms, it is the same 25 ms that went into writing the internal write. As a result, in the asynchronous case, the drive does fewer internal writes, which accounts for the fast completion time. The contiguous placement also explains the 10 ms latency when reading blocks in the written order. Writing synchronously, however, results in doing a separate internal write for every block (shown on the left), taking longer to complete. Placing blocks starting at the beginning of 2.5 track internal writes explains the 5 ms latency when reading blocks in the written order.

To understand how the internal write size changes with the increasing host write size, we keep writing at the maximum queue depth, gradually increasing the write size. Figure 7 shows that the writes in the range of 4 KiB–26 KiB result in 25 ms latency, suggesting that 31 host writes in this size range fit in a single internal write. As we jump to the 28 KiB writes, the latency increases by $\approx$5 ms (or 0.5 track) and remains approximately constant for the writes of sizes up to 54 KiB. We observe a similar jump in latency as we cross from 54 KiB to 56 KiB and also from 82 KiB to 84 KiB. This shows that the internal write size increases in 0.5 track increments. Given that the persistent cache is written using a "log-structured journaling mechanism" [37], we infer that the 0.5 track of 2.5 track minimum internal write is the journal entry that grows in 0.5 track increments, and the remaining 2 tracks contain out-of-band data, like parts of the persistent cache map affected by the host writes. The purpose of this quantization of journal entries is not known, but may be in order to reduce rotational delay or simplify delimiting and locating them. We further hypothesize that the 325 ms delay in Figure 4, observed every 240th write, is a map merge operation that stores the updated map at the middle tracks.

As the write size increases to 256 KiB we see varying delays, and inspection of completion times shows less than 31 writes completing in each burst, implying a bound on the journal entry size. Different completion times for large writes suggest that for these, the journal entry size is determined dynamically, likely based on the available drive resources at the time when the journal entry is formed.

4.4 Disk Cache Location and Layout

We next determine the location and layout of the disk cache, exploiting a phenomenon called *fragmented reads* [20]. When sequentially reading a region in an SMR drive, if the cache contains newer version of some of the blocks in the region, the head has to seek to the persistent cache and back, physically fragmenting a logically sequential read. In Test 2, we use these variations in seek time to discover the location and layout of the disk cache.

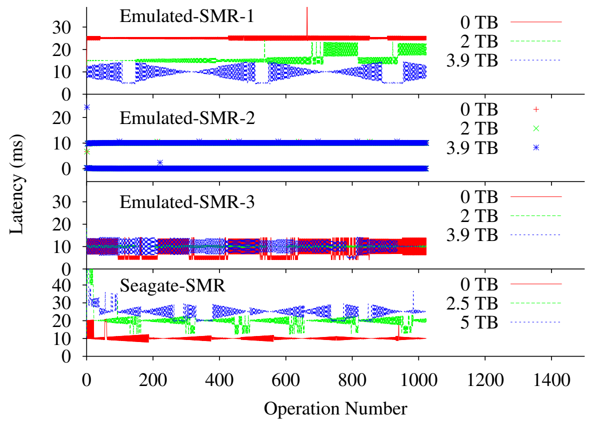


Figure 8: Discovering disk cache structure and location using fragmented reads.

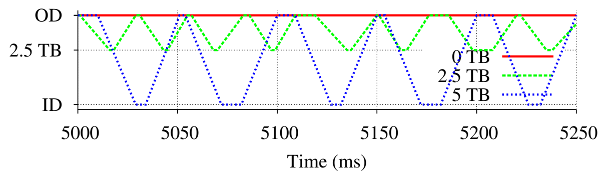


Figure 9: Seagate-SMR head position during fragmented reads.

Test 2: Discovering Disk Cache Location and Layout

```
1 Starting at a given offset, write a block and skip a block, and so on,
  writing 512 blocks in total.
2 Starting at the same offset, read 1024 blocks; call average latency
  $lat_{offset}$.
3 Repeat steps 1 and 2 at the offsets high, low, mid.
4 if $lat_{high} < lat_{mid} < lat_{low}$ then
      There is a single disk cache at the ID.
  else if $lat_{high} > lat_{mid} > lat_{low}$ then
      There is a single disk cache at the OD.
  else if $lat_{high} = lat_{mid} = lat_{low}$ then
      There are multiple disk caches.
  else
      assert($lat_{high} = lat_{low}$ and $lat_{high} > lat_{mid}$)
      There is a single disk cache in the middle.
```

The test works by choosing a small region and writing every other block in it and then reading the region sequentially from the beginning, forcing a fragmented read. LBA numbering conventionally starts at the OD and grows towards the ID. Therefore, a fragmented read at low LBAs on a drive with the disk cache located at the OD would incur negligible seek time, whereas a fragmented read at high LBAs on the same drive would incur high seek time. Conversely, on a drive with the disk cache located at the ID, a fragmented read would incur high seek time at low LBAs and negligible seek time at high LBAs. On a drive with the disk cache located at the middle diameter (MD), fragmented reads at low and high LBAs would incur similar high seek times and they would incur negligible seek times at middle LBAs. Finally, on a drive with multiple disk caches evenly distributed across the drive, the fragmented read latency would be mostly due to rotational delay and vary little across the LBA space. Guided by these assumptions, to identify the location of the disk cache, the test chooses a small region at low, middle, and high LBAs and forces fragmented reads at these regions.

Figure 8 shows the latency of fragmented reads at three offsets on all SMR drives. The test correctly identifies the Emulated-SMR-1 as having a single cache at the ID. For Emulated-SMR-2 with flash cache, latency is seen to be negligible for flash reads, and a full missed rotation for each disk read. Emulated-SMR-3 is also correctly identified as having multiple disk caches—the latency graph of all fragmented reads overlap, all having the same 10 ms average latency. For Seagate-SMR[1] we confirm that it has a single disk cache at OD.

Figure 9 shows the Seagate-SMR head position during fragmented reads at offsets of 0 TB, 2.5 TB and 5 TB. For offsets of 2.5 TB and 5 TB, we see that the head seeks back and forth between the OD and near-center and between the OD and the ID, respectively, occasionally missing a rotation. The cache-to-data distance for LBAs near 0 TB was too small for the resolution of our camera.

4.5 Cleaning

The fragmented read effect is also used in Test 3 to determine whether the drive uses aggressive or lazy cleaning, by creating a fragmented region and then pausing to allow an aggressive cleaning to run before reading the region back.

Test 3: Discovering Cleaning Type

```
1 Starting at a given offset, write a block and skip a block and so on,
  writing 512 blocks in total.
2 Pause for 3–5 seconds.
3 Starting at the same offset, read 1024 blocks.
4 if latency is fixed then cleaning is aggressive else cleaning is lazy.
```

Figure 10 shows the read latency graph of step 3 from Test 3 at an offset of 2.5 TB, with a three second pause in step 2. For all drives, offsets were chosen to land within a single band (§ 4.7). After a pause the top two emulated drives continue to show fragmented read behavior, indicating lazy cleaning, while in Emulated-SMR-3 and Seagate-SMR reads are no longer fragmented, indicating aggressive cleaning.

Figure 11 shows the Seagate-SMR head position during the 3.5 second period starting at the beginning of step 2. Two short seeks from the OD to the ID and back are seen in the first 200 ms; their purpose is not known. The RMW operation for cleaning a band starts at 1,242 ms after the last write, when the head seeks to the band at 2.5 TB offset, reads for 180 ms and seeks back to the cache at the OD where it spends 1,210 ms. We believe this time is spent

[1]Test performed with volatile cache enabled with `hdparm -W1`.

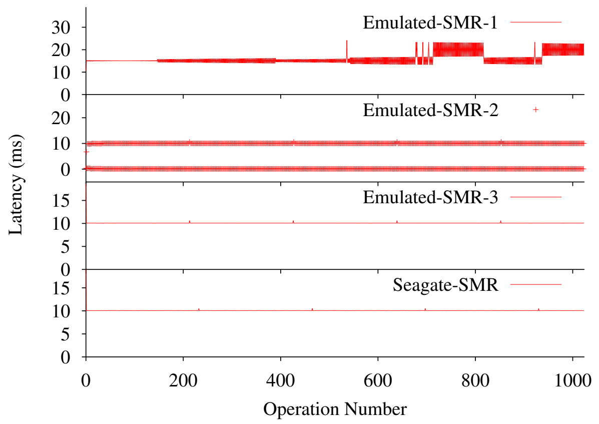


Figure 10: Discovering cleaning type.

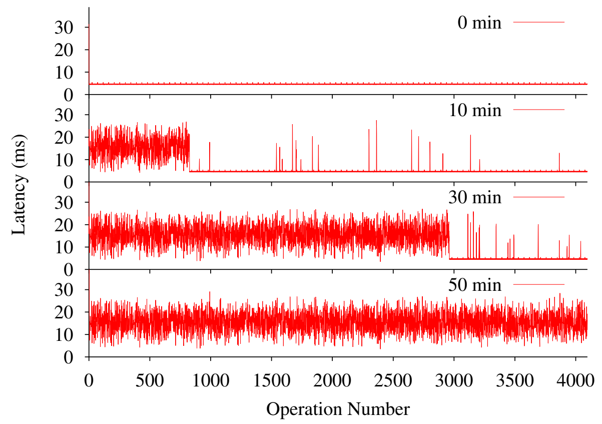


Figure 12: Latency of reads of random writes immediately after the writes and after 10–20 minute pauses.

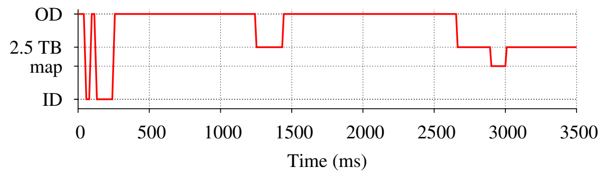


Figure 11: Seagate-SMR head position during the 3.5 second period starting at the beginning of step 2 of Test 3.

forming an updated band and persisting it to the disk cache, to protect against power failure during band overwrite. Next, the head seeks to the band, taking 227 ms to overwrite it and then seeks to the center to update the map. Hence, cleaning a band with half of its content overwritten takes ≈ 1.6 s. We believe the center to contain the map because the head always moves to this position after performing a RMW, and stays there for a short period before eventually parking at the ID. At 3 seconds reads begin and the head seeks back to the band location, where it stays until reads complete (only the first 500 ms is seen in Figure 11).

We confirmed that the operation starting at 1,242 ms is indeed an RMW: when step 3 is begun before the entire cleaning sequence has completed, read behavior is unchanged from Test 2. We did not explore the details of the RMW; alternatives like *partial read-modify-write* [38] may also have been used.

4.5.1 Seagate-SMR Cleaning Algorithm

We next start exploring performance-relevant details that are specific to the Seagate-SMR cleaning algorithm, by running Test 4. In step 1, as the drive receives random writes, it sequentially logs them to the persistent cache as they arrive. Therefore, immediately reading the blocks back in the written order should result in a fixed rotational delay with no seek time. During the pause in step 3, cleaning process moves the blocks from the persistent cache to their native locations. As a result, reading after the pause should incur varying seek time and rotational delay for the blocks moved by the cleaning process, whereas unmoved blocks should still incur a fixed latency.

Test 4: Exploring Cleaning Algorithm

```
1 Write 4096 random blocks.
2 Read back the blocks in the written order.
3 Pause for 10–20 minutes.
4 Repeat steps 2 and 3.
```

In Figure 12 read latency is shown immediately after step 2, and then after 10, 30, and 50 minutes. We observe that the latency is fixed when we read the blocks immediately after the writes. If we re-read the blocks after a 10-minute pause, we observe random latencies for the first ≈ 800 blocks, indicating that the cleaning process has moved these blocks to their native locations. Since every block is expected to be on a different band, the number of operations with random read latencies after each pause shows the progress of the cleaning process, that is, the number of bands it has cleaned. Given that it takes ≈ 30 minutes to clean $\approx 3{,}000$ bands, it takes ≈ 600 ms to clean a band whose single block has been overwritten. We also observe a growing number of cleaned blocks in the unprocessed region (for example, operations 3,000–4,000 in the 30 minute graph); based on this behavior, we hypothesize that cleaning follows Algorithm 1.

Algorithm 1: Hypothesized Cleaning Algorithm of Seagate-SMR

```
1 Read the next block from the persistent cache, find the block's band.
2 Scan the persistent cache identifying blocks belonging to the band.
3 Read-modify-write the band, update the map.
```

To test this hypothesis we run Test 5. In Figure 13 we see that after one minute, all of the blocks written in step 1, some of those written in step 2, and all of those written in step 3 have been cleaned, as indicated by non-uniform

Test 5: Verifying the Hypothesized Cleaning Algorithm

1 Write 128 blocks from a 256 MiB linear region in random order.
2 Write 128 random blocks across the LBA space.
3 Repeat step 1, using different blocks.
4 Pause for one minute; read all blocks in the written order.

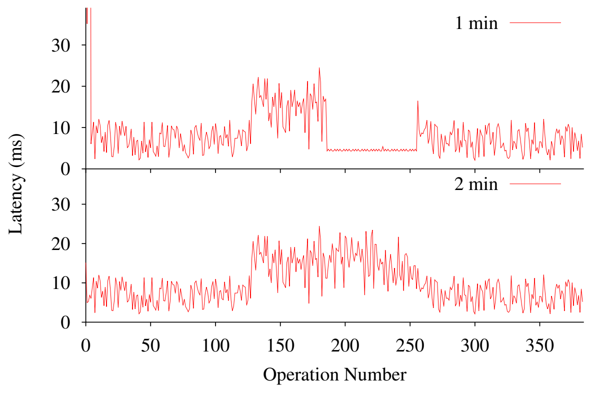


Figure 13: Verifying hypothesized cleaning algorithm on Seagate-SMR.

latency, while the remainder of step 2 blocks remain in cache, confirming our hypothesis. After two minutes all blocks have been cleaned. (The higher latency for step 2 blocks is due to their higher mean seek distance.)

4.6 Persistent Cache Size

We discover the size of the persistent cache by ensuring that the cache is empty and then measuring how much data may be written before cleaning begins. We use random writes across the LBA space to fill the cache, because sequential writes may fill the drive bypassing the cache [20] and cleaning may never start. Also, with sequential writes, a drive with multiple caches may fill only one of the caches and start cleaning before all of the caches are full [20]. With random writes, bypassing the cache is not possible; also, they will fill multiple caches at the same rate and start cleaning when all of the caches are almost full.

The simple task of filling the cache is complicated in drives using extent mapping: a cache is considered full when the extent map is full or when the disk cache is full, whichever happens first. The latter is further complicated by journal entries with quantized sizes—as seen previously (§ 4.3.1), a single 4 KB write may consume as much cache space as dozens of 8 KB writes. Due to this overhead, actual size of the disk cache is larger than what is available to host writes—we differentiate the two by calling them *persistent cache **raw** size* and *persistent cache size*, respectively.

Figure 14 shows three possible scenarios on a hypothetical drive with a persistent cache raw size of 36 blocks and a 12 entry extent map. The minimum journal entry size is 2 blocks, and it grows in units of 2 blocks to the maximum of 16 blocks; out-of-band data of 2 blocks is written with every journal entry; the persistent cache size is 32 blocks.

Part (a) of Figure 14 shows the case of queue depth 1 and 1-block writes. After the host issues 9 writes, the drive puts every write to a separate 2-block journal entry, fills the cache with 9 journal entries and starts cleaning. Every write consumes a slot in the map, shown by the arrows. Due to low queue depth, the drive leaves one empty block in each journal entry, wasting 9 blocks. Exploiting this behavior, Test 6 discovers the persistent cache raw size. In this and the following tests, we detect the start of cleaning by the drop of the IOPS to near zero.

Test 6: Discovering Persistent Cache Raw Size

1 Write with a small size and low queue depth until cleaning starts.
2 Persistent cache raw size = number of writes ×
(minimum journal entry size + out-of-band data size).

Part (b) of Figure 14 shows the case of queue depth 4 and 1-block writes. After the host issues 12 writes, the drive forms three 4-block journal entries. Writing these journal entries to the cache fills the map and the drive starts cleaning despite a half-empty cache. We use Test 7 to discover the *persistent cache map size*.

Test 7: Discovering Persistent Cache Map Size

1 Write with a small size and high queue depth until cleaning starts.
2 Persistent cache map size = number of writes.

Finally, part (c) of Figure 14 shows the case of queue depth 4 and 4-block writes. After the host issues 8 writes, the drive forms two 16-block journal entries, filling the cache. Due to high queue depth and large write size, the drive is able to fill the cache (without wasting any blocks) before the map fills. We use Test 8 to discover the *persistent cache size*.

Test 8: Discovering Persistent Cache Size

1 Write with a large size and high queue depth until cleaning starts.
2 Persistent cache size = total host write size.

Table 2 shows the result of the tests on Seagate-SMR. In the first row, we discover persistent cache raw size using Test 6. Writing with 4 KiB size and queue depth of 1 produces constant 25 ms latency (§ 4.3), that is 2.5 rotations. Track size is ≈ 2 MiB at the OD, therefore, 22,800 operations correspond to ≈ 100 GiB.

In rows 2 and 3 we discover the persistent cache map size using Test 7. For write sizes of 4 KiB and 64 KiB cleaning starts after ≈ 182,200 writes, which corresponds to 0.7 GiB and 11.12 GiB of host writes, respectively. This confirms that in both cases the drive hits the map size limit, corresponding to scenario (b) in Figure 14. Assuming that the drive uses

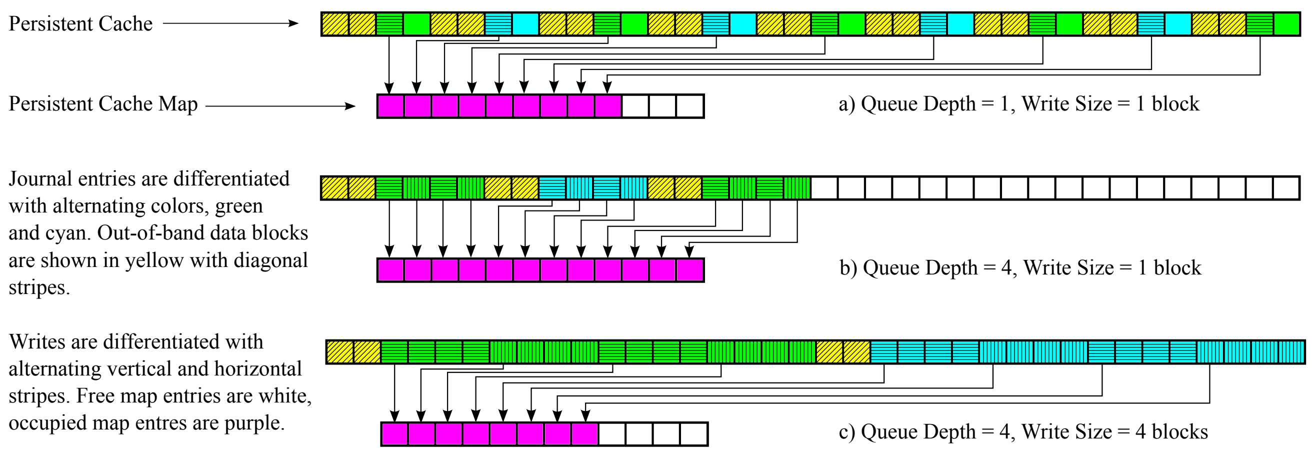

Persistent Cache
Persistent Cache Map
a) Queue Depth = 1, Write Size = 1 block
Journal entries are differentiated with alternating colors, green and cyan. Out-of-band data blocks are shown in yellow with diagonal stripes.
b) Queue Depth = 4, Write Size = 1 block
Writes are differentiated with alternating vertical and horizontal stripes. Free map entries are white, occupied map entres are purple.
c) Queue Depth = 4, Write Size = 4 blocks

Figure 14: Three different scenarios triggering cleaning on drives using journal entries with quantized sizes and extent mapping. The text on the left explains the meaning of the colors.

Drive	Write Size	QD	Operation Count	Host Writes	Internal Writes
	4 KiB	1	22,800	89 MiB	**100 GiB**
	4 KiB	31	**182,270**	0.7 GiB	N/A
Sea-SMR	64 KiB	31	**182,231**	11.12 GiB	N/A
	128 KiB	31	137,496	**16.78 GiB**	N/A
	256 KiB	31	67,830	**16.56 GiB**	N/A
Em-SMR-1	4 KiB	1	9,175,056	35 GiB	35 GiB
Em-SMR-2	4 KiB	1	2,464,153	9.4 GiB	9.4 GiB
Em-SMR-3	4 KiB	1	9,175,056	35 GiB	35 GiB

Table 2: Discovering persistent cache parameters.

a low watermark to trigger cleaning, we estimate that the map size is 200,000 entries.

In rows 4 and 5 we discover the persistent cache size using Test 8. With 128 KiB writes we write $\approx$ 17 GiB in fewer operations than in row 3, indicating that we are hitting the size limit. To confirm this, we increase write size to 256 KiB in row 5; as expected, the number of operations drops by half while the total write size stays the same. Again, assuming that the drive has hit the low watermark, we estimate that the persistent cache size is 20 GiB.

Journal entries with quantized sizes and extent mapping are absent topics in academic literature on SMR, so emulated drives implement neither feature. Running Test 6 on emulated drives produces all three answers, since in these drives, the cache is block-mapped, and the cache size and cache raw size are the same. Furthermore, set-associative STL divides the persistent cache into cache bands and assigns data bands to them using modulo arithmetic. Therefore, despite having a single cache, under random writes it behaves similarly to a fully-associative cache. The bottom rows of Table 2 show that in emulated drives, Test 8 discovers the cache size (see Table 1) with 95% accuracy.

4.7 Band Size

STLs proposed to date [15, 20, 31] clean a single band at a time, by reading unmodified data from a band and updates from the cache, merging them, and writing the merge result back to a band. Test 9 determines the band size, by measuring the granularity at which this cleaning process occurs.

Test 9: Discovering the Band Size

```
1  Select an accuracy granularity $a$, and a band size estimate $b$.
2  Choose a linear region of size $100 \times b$ and divide it into $a$-sized blocks.
3  Write 4 KiB to the beginning of every $a$-sized block, in random order.
4  Force cleaning to run for a few seconds and read 4 KiB from the
   beginning of every $a$-sized block in sequential order.
5  Consecutive reads with identical high latency identify a cleaned band.
```

Assuming that the linear region chosen in Test 9 lies within a region of equal track length, for data that is not in the persistent cache, 4 KB reads at a fixed stride a should see identical latencies—that is, a rotational delay equivalent to $(a \bmod T)$ bytes where T is the track length. Conversely reads of data from cache will see varying delays in the case of a disk cache due to the different (and random) order in which they were written or sub-millisecond delays in the case of a flash cache.

With aggressive cleaning, after pausing to allow the disk to clean a few bands, a linear read of the written blocks will identify the bands that have been cleaned. For a drive with lazy cleaning the linear region is chosen so that writes fill the persistent cache and force a few bands to be cleaned, which again may be detected by a linear read of the written data.

In Figure 15 we see the results of Test 9 for $a = 1$ MiB and $b = 50$ MiB, respectively, with the region located at the 2.5 TB offset; for each drive we zoom in to show an individual band that has been cleaned. We correctly identify the band size for the emulated drives (see Table 1). The band size of Seagate-SMR at this location is seen to be 30 MiB; running tests at different offsets shows that bands are iso-capacity within a zone

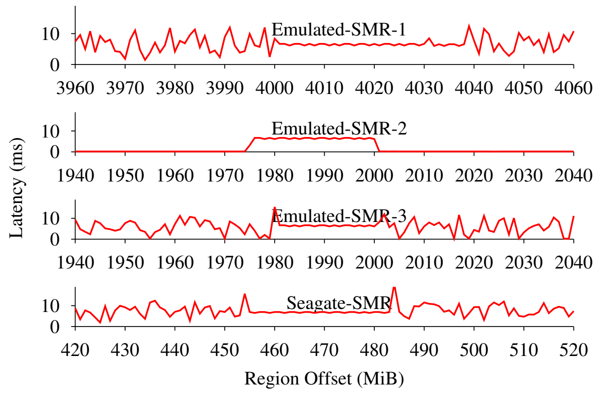


Figure 15: Discovering band size.

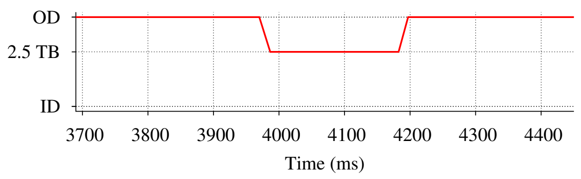


Figure 16: Head position during the sequential read for Seagate-SMR, corresponding to the time period in Figure 15.

(§ 4.9) but vary from 36 MiB at the OD to 17 MiB at the ID.

Figure 16 shows the head position of Seagate-SMR corresponding to the time period in Figure 15. It shows that the head remains at the OD during the reads from the persistent cache up to 454 MiB, then seeks to 2.5 TB offset and stays there for 30 MiB, and then seeks back to the cache at OD, confirming that the blocks in the band are read from their native locations.

4.8 Block Mapping

Once we discover the band size (§ 4.7), we can use Test 10 to determine the mapping type. This test exploits varying inter-track switching latency between different track pairs to detect if a band was remapped. After overwriting the first two tracks of band *b*, cleaning will move the band to its new location—a different physical location only if dynamic mapping is used. Plotting latency graphs of step 2 and step 4 will produce the same pattern for the static mapping and a different pattern for the dynamic mapping.

Adapting this test to a drive with lazy cleaning involves some extra work. First, we should start the test on a drive after a secure erase, so that the persistent cache is empty. Due to lazy cleaning, the graph of step 4 will be the graph of switching between a track and the persistent cache. Therefore, we will fill the cache until cleaning starts, and repeat step 2 once in a while, comparing its graph to the previous two: if it is similar to the last, then data is still in the cache, if it is similar to the first, then the drive uses static mapping, otherwise, the drive uses dynamic mapping.

We used track and block terms to concisely describe the algorithm above, but the size chosen for these algorithmic parameters need not match track size and block size of the underlying drive. Figure 17, for example, shows the plots for the test on Emulated-SMR-3 and Seagate-SMR, using 2 MiB for the track size and 16 KiB for the block size. The latency pattern for the Seagate-SMR does not change, indicating a static mapping, but it changes for Emulated-SMR-3, which indeed uses dynamic mapping. We omit the graphs of the remaining drives to save space.

Test 10: Discovering mapping type.

```
1 Choose two adjacent iso-capacity bands $a$ and $b$; set $n$ to the
  number of blocks in a track.
2 for $i \leftarrow 0$ to $i < 2$ do
      for $j \leftarrow 0$ to $j < n$ do
          Read block $j$ of track 0 of band $a$
          Read block $j$ of track $i$ of band $b$
3 Overwrite the first two tracks of band $b$; force cleaning to run.
4 Repeat step 2.
```

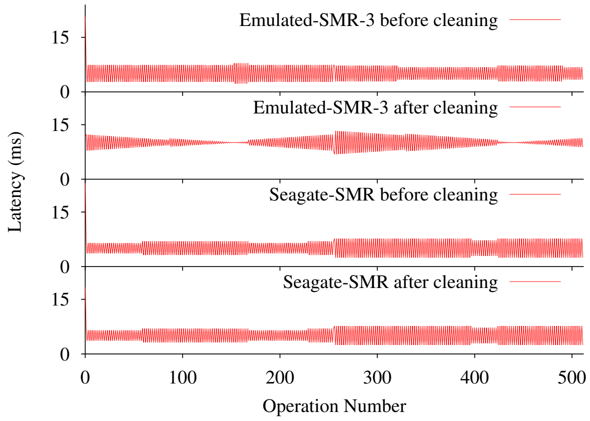


Figure 17: Mapping Type Detection.

4.9 Zone Structure

We use sequential reads (Test 11) to discover the zone structure of Seagate-SMR. While there are no such drives yet, on drives with dynamic mapping a secure erase that would restore the mapping to the default state may be necessary for this test to work. Figure 18 shows the zone profile of Seagate-SMR, with a zoom to the beginning.

Test 11: Discovering Zone Structure

```
1 Enable kernel read-ahead and drive look-ahead.
2 Sequentially read the whole drive in 1 MiB blocks.
```

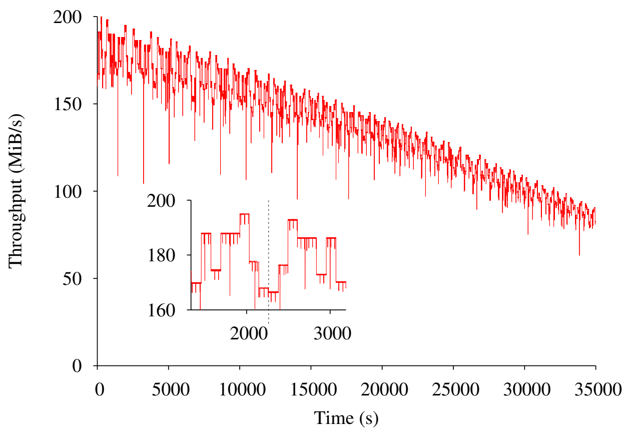


Figure 18: Sequential read throughput of Seagate-SMR.

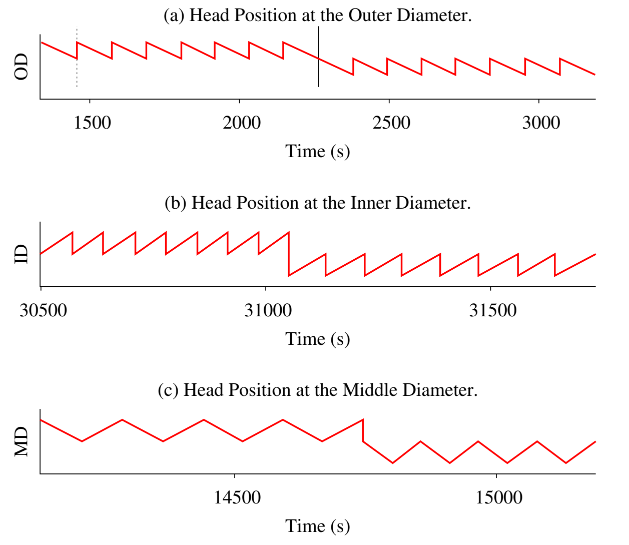


Figure 19: Seagate-SMR head position during sequential reads at different offsets.

Similar to CMR drives, the throughput falls as we reach higher LBAs; unlike CMR drives, there is a pattern that repeats throughout the graph, shown by the zoomed part. This pattern has an axis of symmetry indicated by the dotted vertical line at 2,264th second. There are eight distinct plateaus to the left and to the right of the axis with similar throughputs. The fixed throughput in a single plateau and a sharp change in throughput between plateaus suggest a wide radial stroke and a head switch. Plateaus corresponds to large zones of size 18–20 GiB, gradually decreasing to 4 GiB as we approach higher LBAs. The slight decrease in throughput in symmetric plateaus on the right is due to moving from a larger to smaller radii, where sector per track count decreases; therefore, throughput decreases as well.

We confirmed these hypotheses using the head position graph shown in Figure 19 (a), which corresponds to the time interval of the zoomed graph of Figure 18. Unlike with CMR drives, where we could not observe head switches due to narrow radial strokes, with this SMR drive head switches are visible to an unaided eye. Figure 19 (a) shows that the head starts at the OD and slowly moves towards the MD completing this inwards move at 1,457th second, indicated by the vertical dotted line. At this point, the head has just completed a wide radial stroke reading gigabytes from the top surface of the first platter, and it performs a jump back to the OD and starts a similar stroke on the bottom surface of the first platter. The direction of the head movement indicates that the shingling direction is towards the ID at the OD. The head completes the descent through the platters at 2,264th second—indicated by the vertical solid line—and starts its ascent reading surfaces in the reverse order. These wide radial strokes create "horizontal zones" that consist of thousands of tracks on the same surface, as opposed to "vertical zones" spanning multiple platters in CMR drives. We expect these horizontal zones to be the norm in SMR drives, since they facilitate SMR mechanisms like allocation of iso-capacity bands, static mapping, and dynamic band size adjustment [35]. Figure 19 (b) corresponds to the end of Figure 18, shows that the direction of the head movement is reversed at the ID, indicating that both at the OD and at the ID, shingling direction is towards the middle diameter. To our surprise, Figure 19 (c) shows that a conventional serpentine layout with wide serpents is used at the MD. We speculate that although the whole surface is managed as if it is shingled, there is a large region in the middle that is not shingled.

5 Related Work

Little has been published on the subject of system-level behavior of SMR drives. Although several works (for example, Amer et al. [15] and Le et al. [39]) have discussed requirements and possibilities for use of shingled drives in systems, only three papers to date—Cassuto et al. [20], Lin et al. [40], and Hall et al. [21]—present example translation layers and simulation results. A range of STL approaches is found in the patent literature [27,31,35,41], but evaluation and analysis is lacking. Several SMR-specific file systems have been proposed, such as SMRfs [14], SFS [18], and HiSMRfs [42]. He and Du [43] propose a static mapping to minimize re-writes for in-place updates, which requires high guard overhead (20%) and assumes file system free space is contiguous in the upper LBA region. Pitchumani et al. [32] present an emulator implemented as a Linux device mapper target that mimics shingled writing on top of a CMR drive. Tan et al. [44] describe a simulation of S-blocks algorithm, with a more accurate simulator calibrated with data from a real CMR drive. To date no work (to the authors' knowledge) has presented measurements of read and write operations on an SMR drive, or performance-accurate emulation of STLs.

This work draws heavily on earlier disk characterization

	Drive Model	
Property	ST5000AS0011	ST8000AS0011
Drive Type	SMR	SMR
Persistent Cache Type	Disk	Disk
Cache Layout and Location	Single, at the OD	Single, at the OD
Cache Size	20 GiB	25 GiB
Cache Map Size	200,000	250,000
Band Size	17–36 MiB	15–40 MiB
Block Mapping	Static	Static
Cleaning Type	Aggressive	Aggressive
Cleaning Algorithm	FIFO	FIFO
Cleaning Time	0.6–1.6 s/band	0.6–1.6 s/band
Zone Structure	4–20 GiB	5–40 GiB
Shingling Direction	Towards MD	N/A

Table 3: Properties of the 5 TB and the 8 TB Seagate drives discovered using Skylight methodology. The benchmarks worked out of the box on the 8 TB drive. Since the 8 TB drive was on loan, we did not drill a hole on it; therefore, shingling direction for it is not available.

studies that have used micro-benchmarks to elicit details of internal performance, such as Schlosser et al. [45], Gim et al. [26], Krevat et al. [46], Talagala et al. [25], Worthington et al. [24]. Due to the presence of a translation layer, however, the specific parameters examined in this work (and the micro-benchmarks for measuring them) are different.

6 Conclusions and Recommendations

As Table 3 shows, the Skylight methodology enables us to discover key properties of two drive-managed SMR disks automatically. With manual intervention, it allows us to completely reverse engineer a drive. The purpose of doing so is not just to satisfy our curiosity, however, but to guide both their use and evolution. In particular, we draw the following conclusions from our measurements of the 5 TB Seagate drive:

1. Write latency with the volatile cache disabled is high (Test 1). This appears to be an artifact of specific design choices rather than fundamental requirements, and we hope for it to drop in later firmware revisions.
2. Sequential throughput (with the volatile cache disabled) is much lower (by 3$\times$ or more, depending on write size) than for conventional drives. (We omitted these test results, as performance is identical to the random writes in Test 1.) Due to the use of static mapping (Test 10), achieving full sequential throughput requires enabling volatile cache.
3. Random I/O throughput (with the volatile cache enabled or with high queue depth) is high (Test 7)—15$\times$ that of the equivalent CMR drive. This is a general property of any SMR drive using a persistent cache.
4. Throughput may degrade precipitously when the cache fills after many writes (Table 2). The point at which this occurs depends on write size and queue depth[2].
5. Background cleaning begins after $\approx$1 second of idle time, and proceeds in steps requiring 0.6–1.6 seconds of uninterrupted idle time to clean a single band. The duration of the step depends on the amount of data updated in the band. Cleaning a band whose single block was overwritten may take 0.6 seconds (Figure 12), whereas cleaning a band with half of its content overwritten may take 1.6 seconds (Figure 11). The number of the steps required is proportional to the number of bands—contiguous regions of 15–40 MB (§ 4.7)—that have been modified.
6. Sequential reads of randomly-written data will result in random-like read performance until cleaning completes (§ 4.4).

In summary, SMR drives like the ones we studied should offer good performance if the following conditions are met: (a) the volatile cache is enabled or a high queue depth is used, (b) writes display strong spatial locality, modifying only a few bands at any particular time, (c) non-sequential writes (or all writes, if the volatile cache is disabled) occur in bursts of less than 16 GB or 180,000 operations (Table 2), and (d) long powered-on idle periods are available for background cleaning. From the use of aggressive cleaning that presumes long idle periods, we may conclude that the drive is adapted to desktop use, but may perform poorly on server workloads. Further work will include investigation of STL algorithms that may offer a better balance of performance for both.

Acknowledgments

This work was supported by NetApp, and NSF award CNS-1149232. We thank the anonymous reviewers, Remzi Arpaci-Dusseau, Tim Feldman, and our shepherd, Kimberly Keeton, for their feedback.

References

[1] Seagate Technology PLC Fiscal Fourth Quarter and Year End 2013 Financial Results Supplemental Commentary, July 2013. Available from `http://www.seagate.com/investors`.

[2] Drew Riley. Samsung's SSD Global Summit: Samsung: Flexing Its Dominance In The NAND Market, August 2013.

[3] DRAMeXchange. NAND Flash Spot Price, September 2014. `http://dramexchange.com`.

[2]Although results with the volatile cache enabled are not presented in § 4.6, they are similar to those for a queue depth of 31.

[4] S. N. Piramanayagam. Perpendicular recording media for hard disk drives. *Journal of Applied Physics*, 102(1):011301, July 2007.

[5] Terascale HDD. Data sheet DS1793.1-1306US, Seagate Technology PLC, June 2013.

[6] D.A Thompson and J.S. Best. The future of magnetic data storage techology. *IBM Journal of Research and Development*, 44(3):311–322, May 2000.

[7] R. Wood, Mason Williams, A Kavcic, and Jim Miles. The Feasibility of Magnetic Recording at 10 Terabits Per Square Inch on Conventional Media. *IEEE Transactions on Magnetics*, 45(2):917–923, February 2009.

[8] Seagate Desktop HDD: ST5000DM000, ST4000DM001. Product Manual 100743772, Seagate Technology LLC, December 2013.

[9] Seagate Ships Worlds First 8TB Hard Drives, August 2014. Available from http://www.seagate.com/about/newsroom/.

[10] HGST Unveils Intelligent, Dynamic Storage Solutions To Transform The Data Center, September 2014. Available from http://www.hgst.com/press-room/.

[11] M.H. Kryder, E.C. Gage, T.W. McDaniel, W.A Challener, R.E. Rottmayer, Ganping Ju, Yiao-Tee Hsia, and M.F. Erden. Heat Assisted Magnetic Recording. *Proceedings of the IEEE*, 96(11):1810–1835, November 2008.

[12] Elizabeth A Dobisz, Z.Z. Bandic, Tsai-Wei Wu, and T. Albrecht. Patterned Media: Nanofabrication Challenges of Future Disk Drives. *Proceedings of the IEEE*, 96(11):1836–1846, November 2008.

[13] Sumei Wang, Yao Wang, and R.H. Victora. Shingled Magnetic Recording on Bit Patterned Media at 10 Tb/in^2. *IEEE Transactions on Magnetics*, 49(7):3644–3647, July 2013.

[14] Garth Gibson and Milo Polte. Directions for Shingled-Write and Two-Dimensional Magnetic Recording System Architectures: Synergies with Solid-State Disks. Technical Report CMU-PDL-09-104, CMU Parallel Data Laboratory, May 2009.

[15] Ahmed Amer, Darrell D. E. Long, Ethan L. Miller, Jehan-Francois Paris, and S. J. Thomas Schwarz. Design Issues for a Shingled Write Disk System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–12, Washington, DC, USA, 2010. IEEE Computer Society.

[16] Garth Gibson and Greg Ganger. Principles of Operation for Shingled Disk Devices. Technical Report CMU-PDL-11-107, CMU Parallel Data Laboratory, April 2011.

[17] Tim Feldman and Garth Gibson. Shingled magnetic recording: Areal density increase requires new data management. *USENIX ;login issue*, 38(3), 2013.

[18] D. Le Moal, Z. Bandic, and C. Guyot. Shingled file system host-side management of Shingled Magnetic Recording disks. In *Proceedings of the 2012 IEEE International Conference on Consumer Electronics (ICCE)*, pages 425–426, January 2012.

[19] INCITS T10 Technical Committee. Information technology - Zoned Block Commands (ZBC). Draft Standard T10/BSR INCITS 536, American National Standards Institute, Inc., September 2014. Available from http://www.t10.org/drafts.htm.

[20] Yuval Cassuto, Marco A. A. Sanvido, Cyril Guyot, David R. Hall, and Zvonimir Z. Bandic. Indirection Systems for Shingled-recording Disk Drives. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–14, Washington, DC, USA, 2010. IEEE Computer Society.

[21] David Hall, John H Marcos, and Jonathan D Coker. Data handling algorithms for autonomous shingled magnetic recording hdds. *IEEE Transactions on Magnetics*, 48(5):1777–1781, 2012.

[22] Luc Bouganim, Bjorn Jnsson, and Philippe Bonnet. uFLIP: understanding flash IO patterns. In *Proceedings of the Int'l Conf. on Innovative Data Systems Research (CIDR)*, Asilomar, California, 2009.

[23] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, pages 181–192, New York, NY, USA, 2009. ACM.

[24] Bruce L. Worthington, Gregory R. Ganger, Yale N. Patt, and John Wilkes. On-line Extraction of SCSI Disk Drive Parameters. In *Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '95/PERFORMANCE '95, pages 146–156, New York, NY, USA, 1995. ACM.

[25] Nisha Talagala, Remzi H. Arpaci-Dusseau, and D. Patterson. Microbenchmark-based Extraction of

Local and Global Disk Characteristics. Technical Report UCB/CSD-99-1063, EECS Department, University of California, Berkeley, 1999.

[26] Jongmin Gim and Youjip Won. Extract and infer quickly: Obtaining sector geometry of modern hard disk drives. *ACM Transactions on Storage (TOS)*, 6(2):6:1–6:26, July 2010.

[27] Jonathan Darrel Coker and David Robison Hall. Indirection memory architecture with reduced memory requirements for shingled magnetic recording devices, November 5 2013. US Patent 8,578,122.

[28] Linux Device-Mapper. Device-Mapper Resource Page. https://sourceware.org/dm/, 2001.

[29] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-structured File System. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 1–15, New York, NY, USA, 1991. ACM.

[30] Serial ATA International Organization. Serial ATA Revision 3.1 Specification. Technical report, Serial ATA International Organization, July 2011.

[31] David Robison Hall. Shingle-written magnetic recording (SMR) device with hybrid E-region, April 1 2014. US Patent 8,687,303.

[32] Rekha Pitchumani, Andy Hospodor, Ahmed Amer, Yangwook Kang, Ethan L. Miller, and Darrell D. E. Long. Emulating a Shingled Write Disk. In *Proceedings of the 2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS '12, pages 339–346, Washington, DC, USA, 2012. IEEE Computer Society.

[33] Tim Feldman. Personal communication, August 2014.

[34] Jens Axboe. Flexible I/O Tester. git://git.kernel.dk/fio.git.

[35] Timothy Richard Feldman. Dynamic storage regions, February 14 2011. US Patent App. 13/026,535.

[36] Libata FAQ. https://ata.wiki.kernel.org/index.php/Libata_FAQ.

[37] Tim Feldman. Host-Aware SMR. OpenZFS Developer Summit, November 2014. Available from https://www.youtube.com/watch?v=b1yqjV8qemU.

[38] Sundar Poudyal. Partial write system, March 13 2013. US Patent App. 13/799,827.

[39] Quoc M. Le, Kumar SathyanarayanaRaju, Ahmed Amer, and JoAnne Holliday. Workload Impact on Shingled Write Disks: All-Writes Can Be Alright. In *Proceedings of the 2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '11, pages 444–446, Washington, DC, USA, 2011. IEEE Computer Society.

[40] Chung-I Lin, Dongchul Park, Weiping He, and David H. C. Du. H-SWD: Incorporating Hot Data Identification into Shingled Write Disks. In *Proceedings of the 2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS '12, pages 321–330, Washington, DC, USA, 2012. IEEE Computer Society.

[41] Robert M Fallone and William B Boyle. Data storage device employing a run-length mapping table and a single address mapping table, May 14 2013. US Patent 8,443,167.

[42] Chao Jin, Wei-Ya Xi, Zhi-Yong Ching, Feng Huo, and Chun-Teck Lim. HiSMRfs: A high performance file system for shingled storage array. In *Proceedings of the 2014 IEEE 30th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–6, June 2014.

[43] Weiping He and David H. C. Du. Novel Address Mappings for Shingled Write Disks. In *Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'14, pages 5–5, Berkeley, CA, USA, 2014. USENIX Association.

[44] S. Tan, W. Xi, Z.Y. Ching, C. Jin, and C.T. Lim. Simulation for a Shingled Magnetic Recording Disk. *IEEE Transactions on Magnetics*, 49(6):2677–2681, June 2013.

[45] Steven W. Schlosser, Jiri Schindler, Stratos Papadomanolakis, Minglong Shao, Anastassia Ailamaki, Christos Faloutsos, and Gregory R. Ganger. On Multidimensional Data and Modern Disks. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4*, FAST'05, pages 17–17, Berkeley, CA, USA, 2005. USENIX Association.

[46] Elie Krevat, Joseph Tucek, and Gregory R. Ganger. Disks Are Like Snowflakes: No Two Are Alike. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 14–14, Berkeley, CA, USA, 2011. USENIX Association.

Non-blocking Writes to Files

Daniel Campello[†] *Hector Lopez*[†] *Luis Useche*[◇*] *Ricardo Koller*[‡*] *Raju Rangaswami*[†]
[†]Florida International University [◇]Google Inc. [‡]IBM TJ Watson Research Center

Abstract

Writing data to a page not present in the file-system page cache causes the operating system to synchronously fetch the page into memory first. *Synchronous* page fetch defines both policy (*when*) and mechanism (*how*), and always blocks the writing process. *Non-blocking writes* eliminate such blocking by buffering the written data elsewhere in memory and unblocking the writing process immediately. Subsequent *reads* to the updated page locations are also made non-blocking. This new handling of writes to non-cached pages allow processes to overlap more computation with I/O and improves page fetch I/O throughput by increasing fetch parallelism. Our empirical evaluation demonstrates the potential of non-blocking writes in improving the overall performance of systems with no loss of performance when workloads cannot benefit from it. Across the Filebench write workloads, non-blocking writes improve benchmark throughput by 7X on average (up to 45.4X) when using disk drives and by 2.1X on average (up to 4.2X) when using SSDs. For the SPECsfs2008 benchmark, non-blocking writes decrease overall average latency of NFS operations between 3.5% and 70% and average write latency between 65% and 79%. When replaying the MobiBench file system traces, non-blocking writes decrease average operation latency by 20-60%.

1 Introduction

Caching and buffering file data within the operating system (OS) memory is a key performance optimization that has been prevalent for over four decades [7, 43]. The OS caches file data in units of pages, seamlessly fetching pages into memory from the backing store when necessary as they are read or written to by a process. This basic design has also carried over to networked file systems whereby the client issues page fetches over the network to a remote file server. An undesirable outcome of this design is that processes are blocked by the OS during the page fetch.

While blocking the process for a page fetch cannot be avoided in case of a read to a non-cached page, it can be entirely eliminated in case of writes. The OS could buffer the data written temporarily elsewhere in memory and unblock the process immediately; fetching and updating the page can be performed asynchronously. This decoupling of page write request by the application process from the OS-level page update allows two crucial performance enhancements. First, the process is free to make progress without having to wait for a slow page fetch I/O operation to complete. Second, the parallelism of page fetch operations increases; this improves page fetch throughput since storage devices offer greater performance at higher levels of I/O parallelism.

In this paper, we explore new design alternatives and optimizations for non-blocking writes, address consistency and correctness implications, and present an implementation and evaluation of these ideas. By separating page fetch policy from fetch mechanism, we implement and evaluate two page fetch policies: *asynchronous* and *lazy*, and two page fetch mechanisms: *foreground* and *background*. We also develop *non-blocking reads* to recently written data in non-cached pages.

We implemented non-blocking writes to files in the Linux kernel. Our implementation works seamlessly inside the OS requiring no changes to applications. We integrate the handling of writes to non-cached file data for both local file systems and network file system clients within a common design and implementation framework. And because it builds on a generic design, our implementation provides a starting point for similar implementations in other operating systems.

We evaluated non-blocking writes using several file system workloads. Across Filebench workloads that perform writes, non-blocking writes improve average benchmark throughput by 7X (up to 45.4X) when using disk drives and by 2.1X (up to 4.2X) when using SSDs. For the SPECsfs2008 benchmark workloads, non-blocking writes decrease overall average latency of NFS operations between 3.5% and 70% and average write latency between 65% and 79% across configurations that were obtained by varying the proportion of NFS write operations and NFS read operations. When replaying the MobiBench file system traces, non-blocking writes decrease average operation latency by 20-60%. Finally, the overhead introduced by non-blocking writes is negligible with no loss of performance when workloads cannot benefit from it.

[*]Work done while at Florida International University.

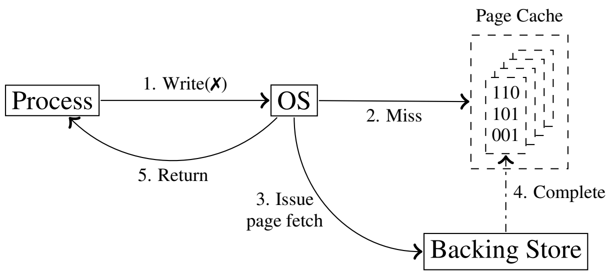


Figure 1: **Anatomy of a write.** The first step, a write reference, fails because the page is not in memory. The process resumes execution (Step 5) only after the blocking I/O operation is completed (Step 4). The dash-dotted arrow represents a slow transition.

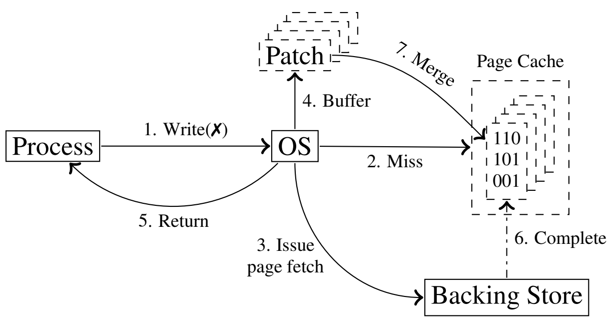


Figure 2: **A non-blocking write employing *asynchronous fetch*.** The process resumes execution (Step 5) after the patch is created in memory while the originally blocking I/O completion is delayed until later (Step 6). The dash-dotted line represents a slow transition.

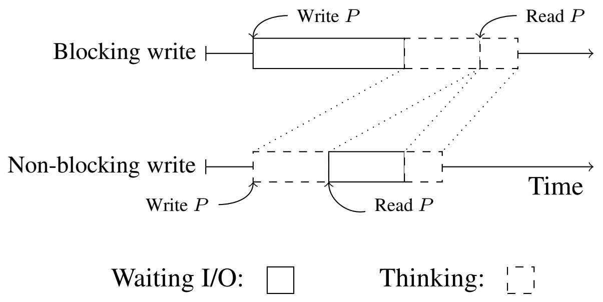


Figure 3: **Page fetch asynchrony with non-blocking writes.** Page P, not present in the page cache, is written to. The application waits for I/O completion. A brief thinktime is followed by a read to P to a different location than the one written to earlier. With non-blocking writes, since the write returns immediately, computation and I/O are performed in parallel.

2 Motivating Non-blocking Writes

Previous studies that have analyzed production file system workloads report a significant fraction of write accesses being small or unaligned writes [11, 30, 39, 44]. Technology trends also indicate an increase in page fetch rates in the future. On the server end, multi-core systems and virtualization now enable more co-located workloads leading to larger memory working sets. As the effective memory working sets [8, 25] of workloads continue to grow, page fetch rates also continue to increase. A host of flash-based hybrid memory systems and storage caching and tiering systems have been inspired, and find relevance in practice, because of these trends [3, 4, 13, 16, 17, 18, 22, 24, 35, 40, 45, 55, 57]. On the personal computing end, newer data intensive desktop/laptop applications place greater I/O demands [20]. In mobile systems, page fetches have been found to affect the performance of the data-intensive applications significantly [23]. Second, emerging byte-addressable persistent memories can provide extremely fast durability to applications and systems software [6, 10, 17, 27, 28, 42, 54, 56, 58]. Recent research has also argued in favor of considering main memory in smartphones as *quasi* non-volatile [32]. When used as file system caches [29, 32], such memories can make the durability of in-memory data a non-blocking operation. Eliminating any unwanted blocking in the front end of the durability process, such as fetch-before-write, becomes critical.

2.1 The *fetch-before-write* problem

Page fetch behavior in file systems is caused because of the mismatch in data access granularities: *bytes* accessed by the application, and *pages* accessed from storage by the operating system. To handle write references, the target page is synchronously fetched before the write is applied, leading to a *fetch-before-write* requirement [34, 51]. This is illustrated in Figure 1. This *blocking* behavior affects performance since it requires fetching data from devices much slower than main memory. Today, main memory accesses can be performed in a couple of nanoseconds whereas accesses to flash drives and hard drives can take hundreds of microseconds to a few milliseconds respectively. We confirmed the page *fetch-before-write* behavior for the latest open-source kernel versions of BSD (all variants), Linux, Minix, OpenSolaris, and Xen.

2.2 Addressing the *fetch-before-write* problem

Non-blocking writes eliminate the *fetch-before-write* requirement by creating an in-memory patch for the updated page and unblocking the process immediately. This modification is illustrated in Figure 2.

2.2.1 Reducing Process blocking

Processes block when they partially overwrite one or more non-cached file pages. Such overwrites may be of any size as long as they are not perfectly aligned to page

Workload	Description
ug-filesrv	Undergrad NFS/CIFS fileserver
gsf-filesrv	Grad/Staff/Faculty NFS/CIFS fileserver
moodle	Web & DB server for department CMS
backup	Nightly backups of department servers
usr1	Researcher 1 desktop
usr2	Researcher 2 desktop
Facebook	MobiBench Facebook trace [14]
twitter	MobiBench twitter trace [14]

Table 1: **Workloads and descriptions.**

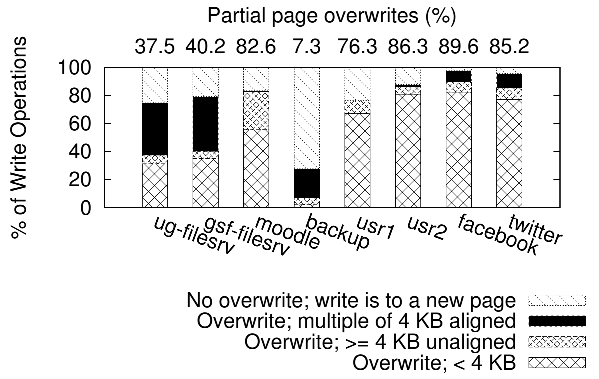


Figure 4: **Breakdown of write operations by amount of page data overwritten.** Each bar represents a different trace and the number above each bar is the percentage of write operations than involve at least one partial page overwrite.

boundaries. Figure 3 illustrates how non-blocking writes reduce process blocking. Previous studies have reported about the significant fraction of small or unaligned writes in production file system workloads [11, 30, 39, 44]. However, little is known about partial page overwrite behavior. To better understand the prevalence of such file writes in production workloads, we developed a Linux kernel module that intercepts file system operations and reports sizes and block alignment for writes. We then analyzed one day's worth of file system operations collected from several production machines at Florida International University's Computer Science department. Besides these we also analyzed file system traces of much shorter duration (two minutes each) available in MobiBench [14, 21]. Table 1 provides a description of all the traces we analyzed.

Figure 4 provides an analysis of the write traffic on each of these machines. On an average, 63.12% of the writes involved partial page overwrites. Depending on the size of the page cache, these overwrites could result in varying degrees of page fetches prior to the page update. The degree of page fetches also depends on the locality of data accesses in the workload wherein a write may follow a read in short temporal order. To account for access locality, we refined our estimates using a cache simulator to count the number of writes that actually lead to page fetches at various memory sizes. Such writes can be made non-blocking. The cache simulator used a modified Mattson's LRU stack algorithm [33] and uses the observation that a non-blocking write at a given LRU cache size would also be a non-blocking write at all smaller cache sizes. Modifications to the original algorithm involved counting all partial page overwrites to pages not in the cache as non-blocking writes. Figure 5 presents the percentage of total writes that would benefit from non-blocking writes for the workloads in Table 1. For a majority of the workloads, this value is at least 15% even for a large page cache of size 100GB. A system that can make such writes non-blocking would make the overall write performance less dependent on the page cache capacity.

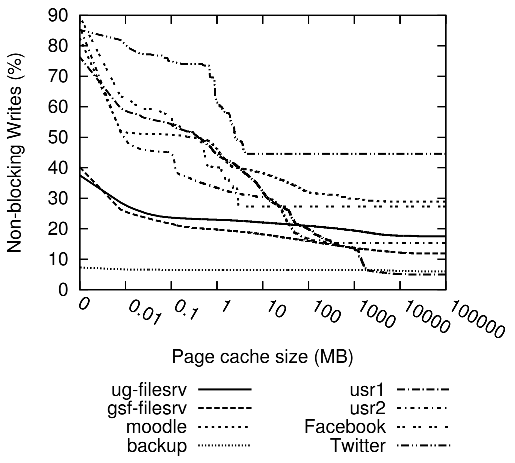


Figure 5: **Non-blocking writes as a percentage of total write operations when varying the page cache size.**

2.2.2 Increasing Page fetch parallelism

Processes that access multiple pages not resident in memory during their execution are blocked by the operating system, once for each page while fetching it. As a result, operating systems end up serializing page fetches for accesses that are independent of each other. With non-blocking writes, the operating system allows a process to fetch independent pages in parallel taking better advantage of the available I/O parallelism at the device level. Figure 6 depicts this improvement graphically. Higher levels of I/O parallelism lead to greater device I/O throughput which ultimately improves page fetch throughput for the application.

2.2.3 Making Durable Writes Fast

Next-generation byte-addressable persistent memories are likely to be relatively small compared to today's block-based persistent stores, at least initially. Main memory in today's smartphones has been argued to be *quasi* non-volatile [32]. When such memories are used

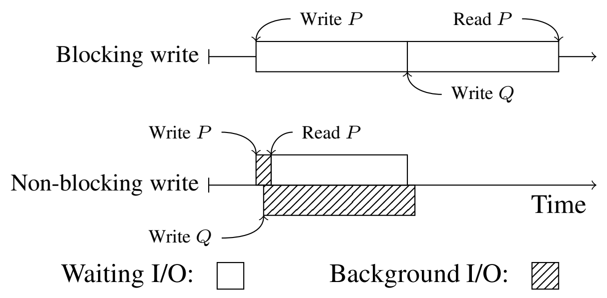


Figure 6: **Page fetch parallelism with non-blocking writes.** Two non-cached pages, P and Q, are written in sequence and the page fetches get serialized by default. With non-blocking writes, P and Q get fetched in parallel increasing device I/O parallelism and thus page fetch throughput.

as a persistent file system cache [29, 32], the containing devices have the ability to provide extremely fast durability (i.e., *sync* operations), a function that would typically block process execution. In such systems, any blocking in the front end of the durability mechanism, such as the fetch-before-write, becomes detrimental to performance. Since non-blocking writes would allow updates without having to fetch the page, it represents the final link in extremely fast data durability when byte addressable persistent memories become widely deployed.

2.3 Addressing Correctness

With non-blocking writes, the ordering of read and write operations within and across processes in the system are liable to change. As we shall elaborate later (§3.3), the *patch creation* and *patch application* mechanisms in non-blocking writes ensure that the ordering of causally dependent operations is preserved. The key insights that we use are: *(i)* reads to recent updates can be served correctly using the most recently created patches, *(ii)* reads that block on page-fetch are allowed to proceed only after applying all the outstanding patches, and *(iii)* reads and writes that are independent and issued by the same or different threads can be reordered without loss of correctness.

Another potential concern with non-blocking writes is data durability. For file data, we observe that the asynchronous write operation only modifies volatile memory and the OS makes no guarantees that the modifications are durable. With non-blocking writes, synchronous writes (on account of `sync/fsync` or the periodic page-flusher daemon) block to wait for the required fetch, apply any outstanding patches, and write the page to storage before unblocking the process. Thus, the durability properties of the system remain unchanged with non-blocking writes.

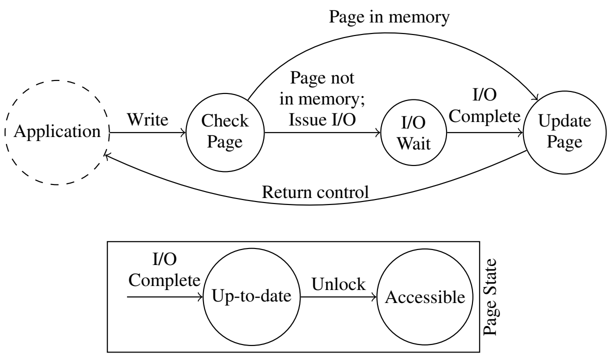


Figure 7: **Process and page state diagram for page fetch with blocking writes.**

3 Non-blocking Writes

The operating system services an application write as depicted in Figure 7. In the *Check Page* state, it looks for the page in the page cache. If the page is already in memory (as a result of a recent fetch completion), it moves to the *Update Page* state which also marks the page as *dirty*. If the page is not in memory, it issues a page fetch I/O and enters the *Wait* state, wherein it waits for the page to be available in memory. When the I/O completes, the page is up-to-date and ready to be unlocked (states *Up-to-date* and *Accessible* in the page state diagram). In the *Update Page* state, the OS makes the page accessible. Finally, control flow returns to the application performing the page write.

3.1 Approach Overview

The page fetch process blocks process execution, which is undesirable. Non-blocking writes work by buffering updates to non-cached pages by creating *patches* in OS memory to be applied later. The basic approach modifies the page fetch path as illustrated in Figure 8. In contrast to current systems, non-blocking writes eliminate the *I/O Wait* state that blocks the process until the page is available in memory. Instead, a non-blocking write returns immediately once a patch of the update is created and queued to the list of pending page updates. non-blocking writes add a new state in the page state, *Outdated*, that reflects the state of the page after it is read into memory but before pending patches are applied. The page transitions into the *Up-to-date* state once all the pending patches are applied.

Non-blocking writes alter write control flow, thus affecting reads to recently written data. Further, they require managing additional cached data in the form of patches. The rest of this section discusses these details in the context of general systems design as well as implementations specific to Linux.

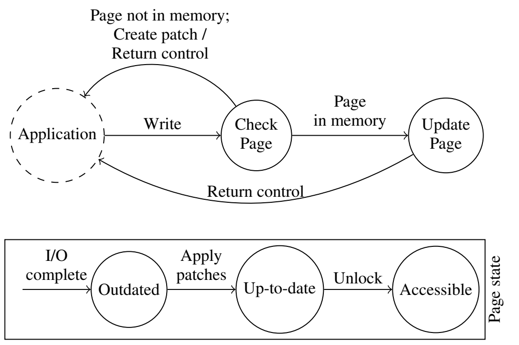


Figure 8: **Process and page state diagram for page fetch with non-blocking writes.**

3.2 Write Handling

Operating systems allow writes to file data via two common mechanisms: supervised *system calls* and unsupervised *memory mapped access*.

To handle supervised writes, the OS uses the system call arguments — the address of the data buffer to be written, the size of the data, and the file (and implicitly, the offset) to write to — and resolves this access to a data page write internally. With non-blocking writes, the OS extracts the data update from the system call arguments, creates a patch, and queues it for later use. This patch is applied later when the data page is read into memory.

Unsupervised file access can be provided by memory mapping a portion of a file to the process address space. In our current design, memory mapped access are handled as in current systems by blocking the process to service the page fault.

3.3 Patch Management

We now discuss how patches are created, stored in the OS, and applied to a page after it is fetched into memory.

3.3.1 Patch Creation

A patch must contain the data to be written along with its target location and size. Since commodity operating systems handle data at the granularity of pages, we chose a design where each patch will apply to a single page. Thus, we abstract an update with a *page patch* data structure that contains all the information to patch and bring the page up-to-date. To handle multiple disjoint overwrites to the same page, we implement *per-page patch queues* wherein page patches are queued and later applied to the page in FIFO order. Consequently, sharing pages via page tables or otherwise is handled correctly. This is possible since operating systems maintain a one-to-one mapping of pages to physical memory frames (e.g., `struct page` in Linux or `struct vm_page` in OpenBSD). When new data is adjacent or overwrites existing patches, it is merged into existing patches accordingly. This makes patch memory overhead and patch application overhead proportional to the number of page bytes changed in the page instead of the number of bytes written to the page since the page was last evicted from memory.

3.3.2 Patch Application

Patch application is rather straightforward. When a page is read in either via a system call induced page fetch or a memory-mapped access causing a page fault, the first step is to apply outstanding patches, if any, to the page to bring it up-to-date before the page is made accessible. Patches are applied by simply copying patch data to the target page location. Patch application occurs in the bottom-half interrupt handling of the page read completion event (further discussed in §5). Once all patches are applied, the page is unlocked which also unblocks the processes waiting on the page, if any.

3.4 Non-blocking Reads

Similar to writes, reads can be classified as *supervised* and *unsupervised* as well. Reads to non-cached pages block the process in current systems. With non-blocking writes, a new opportunity to perform *non-blocking reads* becomes available. Specifically, if the read is serviceable from one of the patches queued on the page, then the reading process can be unblocked immediately without incurring a page fetch I/O. This occurs with no loss of correctness since the patch contains the most recent data written to the page. The read is not serviceable if any portion of the data being requested is not contained within the patch queue. In such a case, the reading process blocks for the page to be fetched. If all data being requested is contained in the patch queue, the data is copied into the target buffer and the reading process is unblocked immediately. For unsupervised reads, our current design blocks the process for the page fetch in all cases.

4 Alternative Page Fetch Modes

Let us consider the page fetch operation issued in Step 3 when performing a non-blocking write as depicted in Figure 2. This operation requires a physical memory allocation (for the page to be fetched) and a subsequent asynchronous I/O to fetch the page so that the newly created patch can be applied to the page. However, since blocking is avoided, process execution is not dependent on the page being available in memory. This raises the question: *can page allocation and fetch be deferred or even eliminated?* Page fetch deferral and elimination allow reduction and shaping of memory consumption and page fetch I/O to storage. While page fetch deferral is opportunistic, page fetch elimination is only possible if the patches created are sufficient to overwrite the page entirely *or* if page persistence becomes unnecessary. We now explore the page fetch modes that become possible with non-blocking writes.

4.1 Asynchronous Page Fetch

In this mode, page fetch I/O is queued to be issued at the time of the page write. The appeal of this approach is its simplicity. Since the page is brought into memory in a timely fashion similar to the synchronous fetch, it is transparent to timer-based durability mechanisms such as dirty page flushing [2] and file system journaling [19].

Asynchronous page fetch defines policy. However, its mechanism may involve additional blocking prior to issuing the page fetch. We discuss two alternative page fetch mechanisms that highlight this issue.

1. **Foreground Asynchronous Page Fetch (NBW-Async-FG)**. The page fetch I/O is issued in the context of process performing the write to the file page. Our discussion in previous sections was based on this mechanism. Although the process does not wait for the completion of the data fetch, issuing the fetch I/O for the data page may itself involve retrieving additional metadata pages to locate the data page if these metadata pages are not cached in OS memory. If so, the writing process would have to block for the necessary metadata fetches to complete, thereby voiding most of the benefits of the non-blocking write.
2. **Background Asynchronous Page Fetch (NBW-Async-BG).** The writing process moves all work necessary to issue the page fetch to a different context by using kernel worker threads. This approach eliminates any blocking of the writing process owing to metadata misses; a worker thread blocks for all fetches while the issuing process continues its execution.

Synchronous fetch is a valuable improvement. However, it consumes system resources, allocating system memory for the page to be fetched and using storage I/O bandwidth to fetch the page.

4.2 Lazy Page Fetch (NBW-Lazy)

When a process writes to a non-cached data page, its execution is not contingent on the page being available in memory. With *lazy page fetch*, the OS delays the page fetch until it becomes unavoidable. Lazy page fetch has the potential to further reduce the system's resource consumption. Figure 9 illustrates this alternative.

Lazy page fetch creates new system scenarios which must be considered carefully. If a future page read cannot be served using the currently available patches for the non-cached page, the page fetch becomes unavoidable. In this case, the page is fetched synchronously and patches are applied first before unblocking the reading process. If the page gets overwritten in its entirety or if page persistence becomes unnecessary for another reason (e.g., the containing file is deleted), the original page fetch is eliminated entirely.

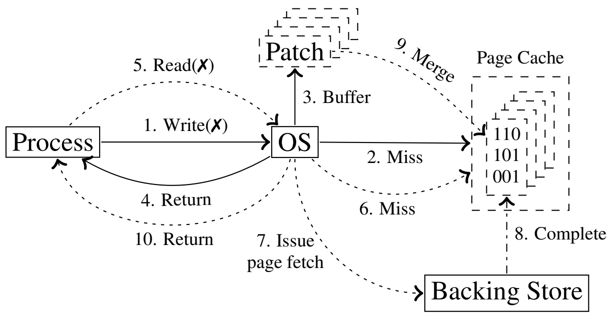


Figure 9: **A non-blocking write employing *lazy fetch*.** The process resumes execution (Step 4) after the patch is created in memory. The Read operation in Step 5 optionally occurs later in the execution while the originally blocking I/O is optionally issued and completes much later (Step 8). The dash-dotted arrow represents a slow transition.

Page data durability can become necessary in the following instances: *(i)* synchronous file write by an application, *(ii)* periodic flushing of dirty pages by the OS [2], or *(iii)* ordered page writes to storage as in a journaling file system [19, 41]. In all these cases, the page is fetched synchronously before being flushed to the backing store. Lastly, non-blocking writes are not engaged for metadata pages which use the conventional durability mechanisms. Durability related questions are discussed further in §5.2.

5 Implementation

Non-blocking writes alter the behavior and control flow of current systems. We present an overview of the implementation of non-blocking writes and discuss details related to how it preserves system correctness.

5.1 Overview

We implemented non-blocking writes for file data in the Linux kernel (version 2.6.34.17) by modifying the generic virtual file system (VFS) layer. Unlike the conventional Linux approach, all handling of fetch completion (such as applying patches, marking the page dirty, processing a journaling transaction, and unlocking the page) occurs in the bottom-half I/O completion handler.

5.2 Handling Correctness

OS-initiated Page Accesses. Our implementation does not implement non-blocking writes for accesses (writes and reads) to un-cached pages initiated internally by the OS. These include file system metadata page updates, and updates performed by kernel threads. This implementation trivially provides the durability properties expected by OS services to preserve semantic correctness.

Journaling File Systems. Our implementation of non-blocking writes preserves the correctness of journaling file systems by allowing the expected behavior for various journaling modes. For instance, non-blocking writes preserve ext4's ordered mode journaling invariant that data updates are flushed to disk before transactions containing related metadata updates. Metadata transactions in ext4 do not get processed until after the related data page is fetched into memory, outstanding patches are applied, the page is marked dirty, and dirty buffers added to the transaction handler. Thus, all dirty data pages related to a metadata transaction are resident in memory and flushed to disk by ext4's ordered mode journaling mechanism prior to committing the transaction.

Handling Read-Write Dependencies. While a non-blocking write is being handled within the operating system, multiple operations such as read, prefetch, synchronous write, and flush, can be issued to the page involved. Operating systems carefully synchronize these operations to maintain consistency and return only up-to-date data to applications. Our implementation respects the Linux page locking protocol. A page is locked after it is allocated and before issuing a fetch for it. As a result, kernel mechanisms such as fsync and mmap are also supported correctly. These mechanisms block on the page lock which becomes available only after the page is fetched and patches applied before proceeding to operate on the page. When delayed page fetch mechanisms (as in NBW-Async-BG and NBW-Lazy) are used, an `NBW` entry for the page involved is added in the page cache mapping for the file before the page is allocated. This `NBW` entry allows for locking the page to maintain the ordering of page operations. When necessary (e.g., a *sync*), pages indexed as `NBW` get fetched which in turn involves acquiring the page lock, thus synchronizing future operations on the page. The only exception to such page locking is writing to a page already in the non-blocking write state; the write does not lock the page but instead queues a new patch.

Ordering of Page Updates. Non-blocking writes may alter the sequence in which patches *to different pages* get applied since the page fetches may complete out-of-order. Non-blocking writes only replace writes that are to memory that are not guaranteed to be reflected to persistent storage in any particular sequence. Thus, ordering violations in updates of in-memory pages are crash-safe.

Page Persistence and Syncs. If an application would like explicit disk ordering for memory page updates, it would execute a blocking flush operation (e.g., `fsync`) subsequent to each operation. The flush operation causes the OS to force the fetch of any page indexed as `NBW` even if it has not been allocated yet. The OS then obtains the page lock, waits for the page fetch, and applies any outstanding patches, before flushing the page and returning control to the application. Ordering of disk writes are thus preserved with non-blocking writes.

Handling of disk errors. Our implementation changes the semantics of the OS with respect to notification of I/O errors when handling writes to non-cached pages. Since page fetches on writes are done asynchronously, disk I/O errors (e.g., `EIO` returned for the UNIX write system call) during the asynchronous page fetch operation would not get reported to the writing application process. Any action that the application was designed to take based on the error reported would not be performed. Semantically, the application write was a memory write and not to persistent storage; an I/O error being reported by current systems is an artifact of the *fetch-before-write* design. With non-blocking writes, if the write were to be made persistent at any point via a flush issued by the application or the OS, any I/O errors during page flushing would be reported to the initiator.

Multi-core and Kernel Preemption. Our implementation fully supports SMP and kernel preemption. For a given non-cached page, the patch creation mechanism (when processing the write system call) can contend with the patch application mechanism (when handling page fetch completion). Our implementation uses a single additional lock to protect a patch queue from simultaneous access.

6 Evaluation

We address the following questions:

(1) What are the benefits of non-blocking writes for different workloads?

(2) How do the fetch modes of non-blocking writes perform relative to each other?

(3) How sensitive are non-blocking writes to the underlying storage type?

(4) How does memory size affect non-blocking writes?

We evaluate four different solutions. Blocking writes (*BW*) is the conventional approach to handling writes and uses the Linux kernel implementation. Non-blocking writes variants include asynchronous mode using foreground (*NBW-Async-FG*) and background (*NBW-Async-BG*) fetch, and lazy mode (*NBW-Lazy*).

Workloads and Experimental Setup. We use the Filebench micro-benchmark [50] to address (1), (2), (3), and (4) using controlled workloads. We use the SPECsfs2008 benchmark [49] and replay the MobiBench traces [14] to further analyze questions (1) and (2); the MobiBench trace replay also helps answer question (3). The Filebench and MobiBench evaluations were performed on a machine with Quad-Core 2.50 GHz AMD Opteron(tm) 1381 processors, 8GB of RAM, a 500

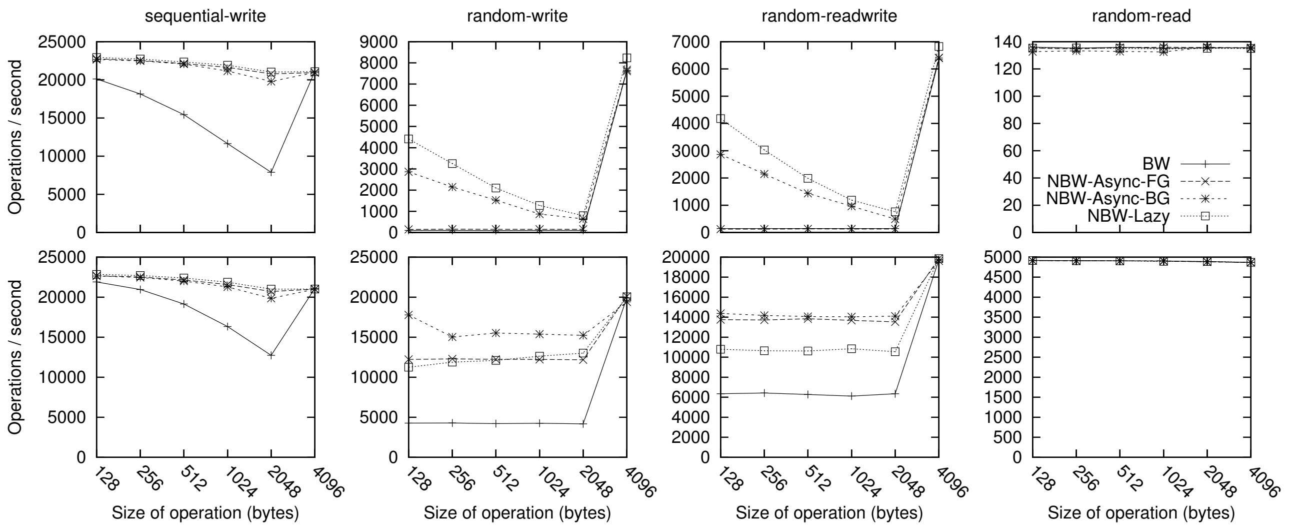


Figure 10: **Performance for various Filebench personalities when varying the I/O size.** The two rows correspond to two different storage back-ends: hard disk-drive (top) and solid-state drive (bottom).

GB WDC WD5002ABYS hard disk-drive, a 32 GB Intel X25-E SSD, and Gigabit Ethernet, running Gentoo Linux (kernel 2.6.34.14) . The above setup was also used to run the client-side component of the SPECsfs2008 benchmark. Additionally, for the SPECsfs2008 benchmark, the NFS server used a 2.3 GHz Quad-Core AMD Opteron(tm) Processor 1356, 7GB of RAM, 500 GB WDC and 160 GB Seagate disks, and Gigabit Ethernet, running Gentoo Linux (kernel 2.6.34.14). The 500GB hard disk housed the root file system while the 160GB hard disk stored the NFS exported data. The network link between client and server was Gigabit Ethernet.

6.1 Filebench Micro-benchmark

For all the following experiments we ran five Filebench personalities for 60 seconds using a 5GB pre-allocated file after clearing the contents of the OS page cache. Each personality represents a different type of workload. The system was configured to use 4GB of main memory and memory used for patches was limited to 64MB, a small fraction of DRAM, to avoid significantly affecting the DRAM available to the workload and the OS. We report the Filebench performance metric, the number of operations per second. Each data-point is calculated using the average of 3 executions.

6.1.1 Performance Evaluation

We first examine the performance of Filebench when using a hard disk as the storage back-end. The top row of Figure 10 depicts the performance for four Filebench personalities when varying the size of the Filebench operation. Each data point reports the average of 3 executions. Standard error of measurement was less than 3% of the average for 96.88% of the cases and were less than 10% for the rest.

The first three plots involve personalities that perform write operations. At 4KB I/O size, there is no *fetch-before-write* behavior because every write results in an overwrite of an entire page; thus, non-blocking writes are not engaged and do not impose any overhead either.

For the *sequential-write* personality, performance with blocking writes (BW) depends on the operation size, and is limited by the number of page misses per operation. In the worst case, when the I/O size is equal to 2KB, every two writes involve a blocking fetch. On average, the different non-blocking write modes provide a performance improvement of 13-160% depending on the I/O size.

The second and third personalities represent random access workloads. *Random-write* is a write-only workload, while *random-readwrite* is a mixed workload; the latter uses two threads, one for issuing reads and the other for writes. For I/O sizes smaller than 4KB, BW provides a constant throughput of around 97 and 146 operations/sec for *random-write* and *random-readwrite* personalities respectively. Performance is consistent regardless of the I/O size because each operation is equally likely to result in a page miss and fetch. *Random-readwrite* performs better than *random-write* due to the additional available I/O parallelism when two threads are used. Further, for *random-write*, NBW-Async-FG provides 50-60% performance improvement due to reduced blocking for page fetches of the process. However, this improvement does not manifest for *random-readwrite* wherein read operations incur higher latencies due to additional blocking for pages with fetches in progress. In both cases, the benefits of NBW-Async-FG are significantly lower when compared to other non-blocking write modes since NBW-Async-FG blocks on many of the ini-

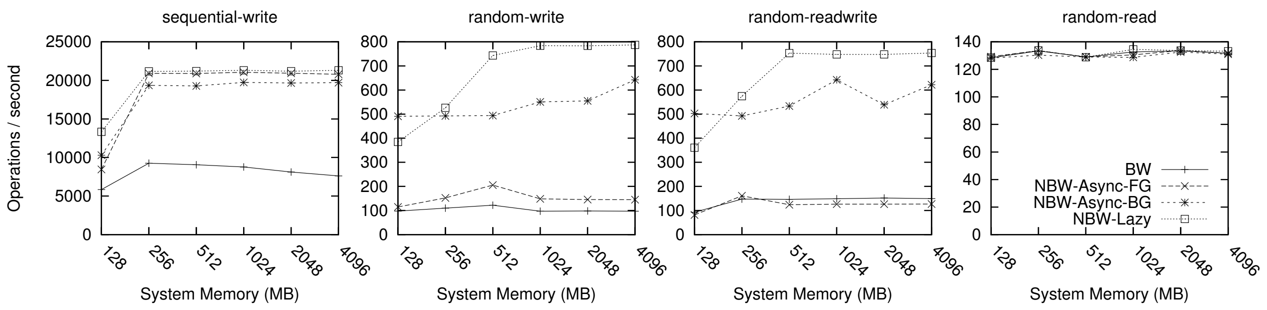


Figure 11: **Memory sensitivity of Filebench.** The I/O size was fixed at 2KB and patch memory limit was set to 64MB.

tial file-system metadata misses during this short-running experiment.

In contrast, NBW-Async-BG unblocks the process immediately while a different kernel thread blocks for the metadata fetches as necessary. This mode shows a 6.7x-29.5x performance improvement for *random-write*, depending on the I/O size. These performance gains reduce as the I/O size increases since non-blocking writes can create fewer outstanding patches to comply with the imposed patch memory limit of 64MB. A similar trend is observed for *random-readwrite* with performance improvements varying from 3.4x-19.5x depending on the I/O size used. NBW-Lazy provides up to 45.4X performance improvement over BW by also eliminating both data and metadata page fetches when possible. When the available patch memory limit is reached, writes are treated as in BW until more patch memory is freed up.

The final two personalities, *random-read* and *sequential-read* (not shown), are read-only workloads. These workloads do not create write operations and the overhead of using a non-blocking writes kernel is zero. Non-blocking writes deliver the same performance as blocking writes.

6.1.2 Sensitivity to system parameters

Our sensitivity analysis of non-blocking writes addresses the following specific questions:

(1) What are the benefits of non-blocking writes when using different storage back-ends?

(2) How do non-blocking writes perform when system memory size is varied?

Sensitivity to storage back-ends

To answer the first question, we evaluated non-blocking writes using a solid state drive (SSD) based storage back-end. Figure 10 (bottom row) presents results when running Filebench personalities using a solid state drive. Each data point reports the average of 3 executions. Standard error of measurement was less than 2.25% of the average in all cases except one for which it was 5%.

Performance trends with the *sequential-write* workload are almost identical to the hard disk counterparts (top row in Figure 10) for all modes of non-blocking writes. This is because non-blocking writes completely eliminate the latency of accessing storage for every operation in both systems. On the other hand, because the SSD offers better throughput than the hard disk drive, BW offers an increase in throughput for every size below 4KB. In summary, the different non-blocking write modes provide between 4% and 61% performance improvement depending on the I/O size.

For the *random-write* and *random-readwrite* workloads, the non-blocking write variants all improve performance but to varying degrees. The SSD had significantly lower latencies servicing random accesses relative to the hard disk drive which allowed for metadata misses to be serviced much quicker. The efficiency of NBW-Async-FG relative to BW is further improved relative to the hard disk system and it delivers 188% and 117% performance improvement for *random-write* and *random-readwrite* respectively. NBW-Async-BG improves over NBW-Async-FG for reasons similar to those with hard disks. NBW-Async-BG delivers 272% (up to 4.2X in the best case) and 125% performance improvement over BW on average for *random-write* and *random-readwrite* respectively. Lastly, although NBW-Lazy performs significantly better than BW, contrary to our expectations, its performance improvements were lower when compared to the NBW-Async modes. Upon further investigation, we found that when the patch memory limit is reached, NBW-Lazy takes longer than the other modes to free its memory given that the fetches are issued only when blocking cannot be avoided anymore. While the duration of the experiment is the same as disk drives, a faster SSD results in the patch memory limit being met more quickly. In our current implementation, after the patch memory limit is reached and no more patches can be created, NBW-Lazy defaults to a BW behavior issuing fetches synchronously for handling writes to non-cached pages. Despite this drawback, NBW-Lazy mode shows 163%-211% and 70% improvement over BW for *random-write* and *random-readwrite* respectively.

Write size	%	Write size	%
1 - 4095 bytes	28	8193 - 16383 bytes	7
4KB	11	16KB	5
4097 - 8191 bytes	3	16385 - 32767 bytes	1
8KB	30	32, 64, 96, 128, 256 KB	15

Table 2: **SPECsfs2008 write sizes.**

Sensitivity to system memory size

We answer the second question using the Filebench workloads and varying the amount of system memory available to the operating system. For these experiments, we used a hard disk drive as the storage back-end and fixed the I/O size at 2KB. Figure 11 presents the results of this experiment. Each data point reports the average of 3 executions. Standard error of measurement was less than 4% of the average for 90% of the cases and were less than 10% for the rest.

For the *sequential-write* workload, the non-blocking writes variants perform 45-180% better than BW. Further NBW-Lazy performs better and can be considered optimal because *(i)* it uses very little patch memory, sufficient to hold enough patches until a single whole page is overwritten, and *(ii)* since pages get overwritten entirely in the sequential write, it eliminates all page fetches.

For *random-write* and *random-readwrite* workloads, NBW-Async-FG delivers performance that is relatively consistent with BW; the I/O performance achieved by these solutions is not high enough to make differences in memory relevant. NBW-Async-BG and NBW-Lazy offer significant performance gains relative to BW of as much as 560% and 710% respectively. With NBW-Lazy, performance improves with more available memory but only up to the point at which the imposed patch memory limit is reached prior to the completion of the execution; increasing the patch memory limit would allow NBW-Lazy to continue scaling its performance.

6.2 SPECsfs2008 Macro-benchmark

The SPECsfs2008 benchmark tests the performance of NFS servers. For this experiment, we installed a non-blocking writes kernel in the NFS server which exported the network file system in *async* mode. SPECsfs2008 uses a client side workload generator that bypasses the page cache entirely. The client was configured for a target load of 500 operations per second. The target load was sustained in all evaluations; thus the SPECsfs2008 performance metric is the operation latency reported by the NFS client. While the evaluation results are encouraging, the relative performance results we report for NFS workloads are likely to be an underestimate. This is because our prototype was used only at the NFS server; the client counterpart of non-blocking writes was not engaged by this benchmark.

SPECsfs2008 operations are classified as *write, read,* and *others* which includes metadata operations such as *create, remove,* and *getattr*. For each variant solution, we report results for the above three classes of operations separately as well as the overall performance that represents the weighted average across all operations. Further, we evaluated performance when varying the relative proportion of NFS operations issued by the benchmark. The default configuration as specified in SPECsfs2008 is: reads (18%), writes (10%) and others (72%). We also evaluated three modified configurations: no-writes, no-reads, and one that uses: reads (10%), writes (18%), and others (72%) to examine a wider spectrum of behaviors.

We first perform a brief analysis of the workload to determine expected performance. Even for configurations that contained more writes than reads (e.g., 18% writes and 10% reads) the actual fraction of cache misses upon writes is far lower than the fraction of misses due to reads (i.e 16.9% write misses vs. 83.1% read misses). This mismatch is explained by noting that each read access to a non-cached page results in a read miss but the same is not true for write accesses when they are page-aligned. Further, Table 2 reports that only 39% of all writes issued by the SPECsfs2008 are partial page overwrites which may result in non-blocking writes.

Figure 12 presents the average operation latencies normalized using the latency with the BW solution. Excluding the read-only workload, the dominant trend is that the non-blocking write modes offer significant reductions in write operation latency with little or no degradation in read latencies. Further, the average overall operation latency is proportional to the fraction of write misses and to the latency improvements for NFS write operations. For the three configurations containing write operations, the latency of the write operations is reduced between 65 and 79 percent when using the different modes of non-blocking writes. Read latencies are slightly affected negatively due to additional blocking on certain pages. With BW, certain pages could have been fetched into memory by the time the read operation was issued. With non-blocking writes, the corresponding fetches could be delayed or not issued at all until the blocking read occurs. For the configuration with no write operations the average overall latency remained relatively unaffected.

6.3 MobiBench Trace Replay

The MobiBench suite of tools contains traces obtained from an Android device when using the Facebook and Twitter apps [14]. We used MobiBench's timing-accurate replay tool to replay the traces. We fixed a bug in the replay tool prior to using it; the original replayer used a fixed set of flags when opening files regardless of the trace information. MobiBench reports the average file system call operation latency as the performance metric. We replayed the traces five times and report the

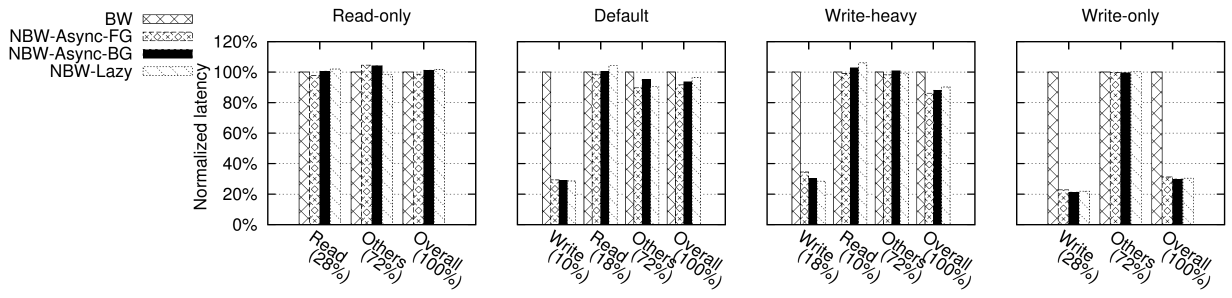


Figure 12: **Normalized average operation latencies for SPECsfs2008.**

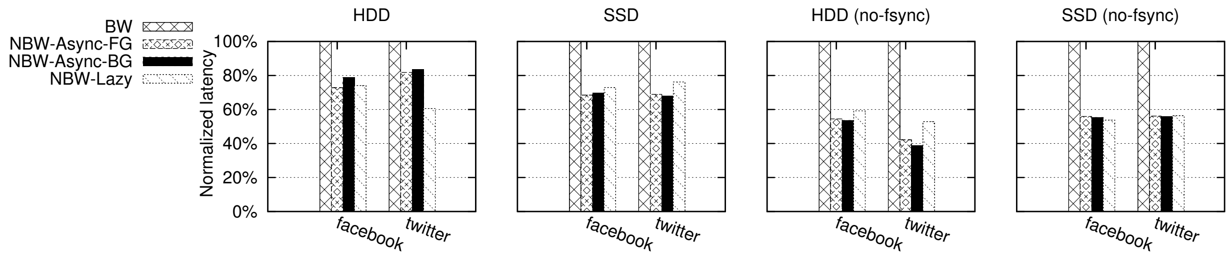


Figure 13: **Normalized average operation latencies when replaying MobiBench traces [14].**

average latency observed. Standard error of measurement was less than 4% of the average in all cases except one for which it was 7.18%. The two left-most graphs of Figure 13 present the results for this evaluation for both hard disks and solid-state drives respectively. Non-blocking writes exhibit a reduction in operation latencies between 20% and 40% depending on the mode and back-end storage used for both Facebook and Twitter traces.

When we analyzed the MobiBench traces, we found that they contained a significant amount of sync operations. Sync operations do not allow exploiting the full potential of non-blocking writes because they block the process to fetch pages synchronously. As discussed previously, recent work on byte-addressable persistent memories and qNVRAM [32] provide for extremely fast, durable, in-memory operations. In such systems, the blocking fetch-before-write behavior in OSes becomes an even more significant obstacle to performance. To estimate[1] the impact of non-blocking writes in such an environment, we modified the original traces by discarding all *fsync* operations to simulate extremely fast durability of in-memory data. The rightmost two graphs present the results obtained upon replaying the modified traces. non-blocking writes reduce latencies by 40-60% depending on the mode and the storage back-end used.

7 Related Work

Non-blocking writes have existed for almost three decades for managing CPU caches. Observing that entire cache lines do not need to be fetched on a word write-miss thereby stalling the processor, the use of additional registers that temporarily store these word updates was investigated [26] and later adopted [31].

Recently, non-blocking writes to main memory pages was motivated using full system memory access traces generated by an instrumented QEMU machine emulator [53]. This prior work outlined some of the challenges of implementing non-blocking writes in commodity operating systems. We improve upon this work by presenting a detailed design and Linux kernel implementation of non-blocking writes, addressing a host of challenges as well as uncovering new design points. We also present a comprehensive evaluation with a wider range of workloads and performance numbers from a running system.

A candidate approach to mitigate the *fetch-before-write* problem involves provisioning adequate DRAM to minimize write cache misses. However, the file system footprint of a workload over time is usually unpredictable and potentially unbounded. Alternatively, prefetching [46] can reduce blocking by anticipating future memory accesses. However, prefetching is typically limited to sequential accesses. Moreover, incorrect decisions can render prefetching ineffective and pollute memory. Non-blocking writes is complementary to these approaches. It uses memory judiciously and only fetches those pages that are necessary for process execution.

There are several approaches proposed in the literature that reduce process blocking specifically for system call induced page fetches. The goal of the asynchronous I/O library (e.g., POSIX AIO [1]) available on Linux and a few BSD variants is to make file system writes asynchronous; a helper library thread blocks on behalf of the

[1]We did not enforce ordered CPU cache flushing to persistent memory to ensure in-memory durability upon fsync.

process. LAIO [12] is a generalization of the basic AIO technique to make all system calls asynchronous; a library checkpoints execution state and relies on scheduler activations to get notified about the completion of blocking I/O operations initiated inside the kernel. Recently, FlexSC [48] proposed asynchronous exceptionless system calls wherein system calls are queued by the process in a page shared between user and kernel space; these calls are serviced asynchronously by syscall kernel threads which report completion back to the user process.

The scope of non-blocking writes in relation to the above proposals is different. Its goal is to entirely eliminate the blocking of memory writes to pages not available in the file system page cache. A non-blocking write does not need to checkpoint state thereby consuming lesser system resources. Further, it can be configured to be lightweight so that it does not use additional threads (often a limited resource in systems) to block on behalf of the running process. Finally, unlike these approaches which require application modifications to use specific libraries, non-blocking writes work seamlessly in the OS transparent to applications.

There are works that are related to non-blocking writes, but quite different in their accomplished goal. Speculative execution (or Speculator) as proposed by Nightingale *et al.* [36] eliminates blocking when synchronously writing cached in-memory page modifications to a network file server using a process checkpoint and rollback mechanism. Xsyncfs [37] eliminates the blocking upon performing synchronous writes of in-memory pages to disk by creating a commit dependency for the write and allowing the process to make progress. Featherstitch [15] improves the performance of synchronous file system page updates by scheduling these page writes to disk more intelligently. Featherstitch employed patches but for a different purpose – to specify dependent changes across disk blocks at the byte granularity. OptFS [5] decouples the ordering of writes of in-memory pages from their durability, thus improving performance. While these approaches optimize the writing of in-memory pages to disk they do not eliminate the blocking page fetch before in-memory modifications to a file page can be made.

BOSC [47] describes a new disk update interface for applications to explicitly specify disk update requests and associate call back functions. Opportunistic Log [38] describes the fetch-before-write problem for objects and uses a second log to record updates. Both of these reduce application blocking allowing updates to happen in the background but they require application modification and do not support general-purpose usage. Non-blocking writes is complementary to the above body of work because it runs seamlessly inside the OS requiring no changes to applications.

8 Conclusions and Future Work

For over four decades, operating systems have blocked processes for page fetch I/O when they write to non-cached file data. In this paper, we revisited this well-established design and demonstrated that such blocking is not just unnecessary but also detrimental to performance. Non-blocking writes decouple the writing of data to a page from its presence in memory by buffering page updates elsewhere in OS memory. This decoupling is achieved with a self-contained operating system improvement seamless to the applications. We designed and implemented *asynchronous* and *lazy* page fetch modes that are worthwhile alternatives to blocking page fetch. Our evaluation of non-blocking writes using Filebench revealed throughput performance improvements of as much as 45.4X across various workload types relative to blocking writes. For the SPECsfs2008 benchmark, non-blocking writes reduced write operation latencies by as much as 65-79%. When replaying the MobiBench file system traces, non-blocking writes decreased average operation latency by 20-60%. Further, there is no loss of performance when workloads cannot benefit from non-blocking writes.

Non-blocking writes open up several avenues for future work. First, since they alter the relative importance of pages in memory in a fundamental way, new page replacement algorithms are worth investigating. Second, by intelligently scheduling page fetch operations (instead of simply asynchronously or lazily), we can reduce and shape both memory consumption and the page fetch I/O traffic to storage. Third, I/O related to asynchronous page fetching due to non-blocking writes can be scheduled more intelligently (e.g., as background operations [52] or semi-preemptibly [9]) to speed up blocking page fetches. Finally, certain OS mechanisms such as dirty page flushing thresholds and limits on per-process dirty data would need to be updated to also account for in-memory patches.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Donald Porter, for their detailed and thoughtful feedback which improved the quality of this paper significantly. Many thanks to Eric Johnson who aided in the collection of file system traces from FIU's production servers. This work is supported in part by NSF awards CNS-1018262 and CNS-1448747, the Intel ISRA program, and a NetApp Faculty Fellowship.

Traces

The traces used in this paper are available at: `http://sylab-srv.cs.fiu.edu/dokuwiki/doku.php?id=projects:nbw:start`

References

[1] American National Standards Institute. *IEEE standard for information technology: Portable Operating Sytem Interface (POSIX). Part 1, system application program interface (API) — amendment 1 — realtime extension [C language]*. IEEE, 1994. IEEE Std 1003.1b-1993 (formerly known as IEEE P1003.4; includes IEEE Std 1003.1-1990). Approved September 15, 1993, IEEE Standards Board. Approved April 14, 1994, American National Standards Institute.

[2] Bach, M. J. *The Design of the UNIX Operating System*, 1st ed. Prentice Hall Press, 1986.

[3] Byan, S., Lentini, J., Madan, A., Pabon, L., Condict, M., Kimmel, J., Kleiman, S., Small, C., and Storer, M. Mercury: Host-side flash caching for the data center. In *Proceedings of the 28th IEEE Conference on Massive Data Storage* (April 2012), MSST '12.

[4] Chen, F., Koufaty, D. A., and Zhang, X. Hystor: making the best use of solid state drives in high performance storage systems. In *Proceedings of the International Conference on Supercomputing* (May-June 2011), ICS '11.

[5] Chidambaram, V., Pillai, T. S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Optimistic crash consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (November 2013), SOSP '13.

[6] Coburn, J., Caulfield, A., Akel, A., Grupp, L., Gupta, R., Jhala, R., and Swanson, S. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (2011), ASPLOS XVI.

[7] Daley, R. C., and Neumann, P. G. A general-purpose file system for secondary storage. In *Proceedings of the Fall Joint Computer Conference, Part I* (1965), AFIPS '65 (Fall, part I), pp. 213–229.

[8] Denning, P. J. The working set model for program behavior. *Communications of the ACM 11*, 5 (1968), 323–333.

[9] Dimitrijevic, Z., Rangaswami, R., and Chang, E. Systems support for preemptive RAID scheduling. *IEEE Transactions on Computers 54*, 10 (October 2005), 1314–1326.

[10] Dulloor, S. R., Kumar, S., Keshavamurthy, A., Lantz, P., Reddy, D., Sankaran, R., and Jackson, J. System software for persistent memory. In *Proceedings of the European Conference on Computer Systems* (2014), EuroSys '14.

[11] Ellard, D., Ledlie, J., Malkani, P., and Seltzer, M. Passive NFS tracing of email and research workloads. In *Proceedings of the USENIX Conference on File and Storage Technologies* (2003), FAST '03.

[12] Elmeleegy, K., Chanda, A., Cox, A. L., and Zwaenepoel, W. Lazy asynchronous I/O for event-driven servers. In *Proceedings of the 2004 USENIX Annual Technical Conference* (2004), ATC '04.

[13] EMC. VFCache. http://www.emc.com/storage/vfcache/vfcache.htm, 2012.

[14] ESOS Laboratory. Mobibench traces. https://github.com/ESOS-Lab/Mobibench/tree/master/MobiGen, 2013.

[15] Frost, C., Mammarella, M., Kohler, E., de los Reyes, A., Hovsepian, S., Matsuoka, A., and Zhang, L. Generalized file system dependencies. In *Proceedings of the ACM Symposium on Operating Systems Principles* (October 2007), SOSP '07, pp. 307–320.

[16] Fusion-IO. ioTurbine. http://www.fusionio.com/systems/ioturbine/, 2012.

[17] Guerra, J., Marmol, L., Campello, D., Crespo, C., Rangaswami, R., and Wei, J. Software persistent memory. In *Proceedings of the USENIX Annual Technical Conference* (June 2012), ATC '12.

[18] Guerra, J., Pucha, H., Glider, J., Belluomini, W., and Rangaswami, R. Cost effective storage using extent-based dynamic tiering. In *Proceedings of the USENIX Conference on File and Storage Technologies* (February 2011), FAST '11.

[19] Hagmann, R. Reimplementing the Cedar file system using logging and group commit. In *Proceedings of the ACM Symposium on Operating Systems Principles* (November 1987), SOSP '87.

[20] Harter, T., Dragga, C., Vaughn, M., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. A file is not a file: Understanding

the I/O behavior of Apple desktop applications. In *Proceedings of the ACM Symposium on Operating Systems Principles* (October 2011), SOSP '11.

[21] JEONG, S., LEE, K., HWANG, J., LEE, S., AND WON, Y. Framework for analyzing android I/O stack behavior: From generating the workload to analyzing the trace. *Future Internet 5*, 4 (2013), 591–610.

[22] KGIL, T., AND MUDGE, T. FlashCache: a NAND flash memory file cache for low power web servers. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems* (October 2006), CASES '06.

[23] KIM, H., AGRAWAL, N., AND UNGUREANU, C. Revisiting storage for smartphones. In *Proceedings of the USENIX Conference on File and Storage Technologies* (February 2012), FAST '12.

[24] KOLLER, R., MARMOL, L., RANGASWAMI, R., SUNDARARAMAN, S., TALAGALA, N., AND ZHAO, M. Write policies for host-side flash caches. In *Proceedings of the USENIX Conference on File and Storage Technologies* (February 2013), FAST '13.

[25] KOLLER, R., VERMA, A., AND RANGASWAMI, R. Generalized ERSS tree model: Revisiting working sets. *Performance Evaluation 67*, 11 (2010), 1139–1154.

[26] KROFT, D. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th annual symposium on Computer Architecture* (1981), ISCA '81, IEEE Computer Society Press, pp. 81–87.

[27] LANTZ, P., DULLOOR, S., KUMAR, S., SANKARAN, R., AND JACKSON, J. Yat: A validation framework for persistent memory software. In *Proceedings of the USENIX Annual Technical Conference* (June 2014), ATC '14.

[28] LEE, B., IPEK, E., MUTLU, O., AND BURGER, D. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the International Symposium on Computer Architecture* (2009), ISCA '09.

[29] LEE, E., BAHN, H., AND NOH, S. H. Unioning of the buffer cache and journaling layers with non-volatile memory. In *Proceedings of the USENIX Conference on File and Storage Technologies* (February 2013), FAST '13.

[30] LEUNG, A. W., PASUPATHY, S., GOODSON, G., AND MILLER, E. L. Measurement and analysis of large-scale network file systemworkloads. In *Proceedings of the USENIX Annual Technical Conference* (2008).

[31] LI, S., CHEN, K., BROCKMAN, J. B., AND JOUPPI, N. P. Performance impacts of non-blocking caches in out-of-order processors. Tech. rep., Hewlett-Packard Labs and University of Notre Dame, July 2011.

[32] LUO, H., TIAN, L., AND JIANG, H. qNVRAM: quasi non-volatile RAM for low overhead persistency enforcement in smartphones. In *6th USENIX Workshop on Hot Topics in Storage and File Systems* (June 2014), HotStorage '14.

[33] MATTSON, R. L., GECSEI, J., SLUTZ, D. R., AND TRAIGER, I. L. Evaluation techniques for storage hierarchies. *IBM Systems Journal* (1970).

[34] MCKUSICK, M. K., BOSTIC, K., KARELS, M. J., AND QUARTERMAN, J. S. *The Design and Implementation of the 4.4 BSD Operating System.* Addison Wesley, 1996, pp. 163, 196.

[35] NETAPP. Flash Accel. http://www.netapp.com/us/products/storage-systems/flash-accel/, 2013.

[36] NIGHTINGALE, E. B., CHEN, P. M., AND FLINN, J. Speculative execution in a distributed file system. *ACM Transactions on Computer Systems* (2006), 361–392.

[37] NIGHTINGALE, E. B., VEERARAGHAVAN, K., CHEN, P. M., AND FLINN, J. Rethink the sync. In *Proceedings of the 7th USENIX Conference on Operating Systems Design and Implementation* (November 2006), OSDI '06.

[38] O'TOOLE, J., AND SHRIRA, L. Opportunistic log: Efficient installation reads in a reliable storage server. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation* (1994), OSDI '94, pp. 39–48.

[39] OUSTERHOUT, J. K., DA COSTA, H., HARRISON, D., KUNZE, J. A., KUPFER, M., AND THOMPSON, J. G. A trace-driven analysis of the UNIX 4.2 BSD/ file system. In *Proceedings of the ACM Symposium on Operating Systems Principles* (1985), SOSP '85.

[40] PATIL, M., VILAYANNUR, M., OSTROWSKI, M., NARKHEDE, S., KOTHAKOTA, V., JUNG, W., KOHLER, H., SOUNDARARAJAN, G., PATIL, K.,

KUMAR, C., AND BHAGWAT, D. A practical implementation of clustered fault tolerant write acceleration in a virtualized environment. In *13th USENIX Conference on File and Storage Technologies* (2015), FAST '15.

[41] PRABHAKARAN, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Analysis and evolution of journaling file systems. In *Proceedings of the USENIX Annual Technical Conference* (June 2005), ATC '05.

[42] QURESHI, M. K., SRINIVASAN, V., AND RIVERS, J. A. Scalable high-performance main memory system using phase-change memory technology. In *Proceedings of the International Symposium on Computer Architecture* (2009), ISCA '09.

[43] RITCHIE, D. M., AND THOMPSON, K. The UNIX time-sharing system. *Commun. ACM 17* (July 1974), 365–375.

[44] ROSELLI, D., AND ANDERSON, T. E. A comparison of file system workloads. In *Proceedings of the USENIX Annual Technical Conference* (2000), ATC '00.

[45] SAXENA, M., AND SWIFT, M. M. FlashVM: Revisiting the virtual memory hierarchy. In *Proceedings of the USENIX Annual Technical Conference* (June 2010), ATC '10.

[46] SHRIVER, E., SMALL, C., AND SMITH, K. A. Why does file system prefetching work? In *Proceedings of the USENIX Annual Technical Conference* (1999), ATC '99.

[47] SIMHA, D. N., LU, M., AND CHIUEH, T.-C. An update-aware storage system for low-locality update-intensive workloads. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2012), ASPLOS XVII.

[48] SOARES, L., AND STUMM, M. FlexSC: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation* (2010), OSDI'10, USENIX Association, pp. 1–8.

[49] STANDARD PERFORMANCE EVALUATION CORPORATION. SPECsfs2008. http://www.spec.org/sfs2008/, 2008.

[50] SUN MICROSYSTEMS. Filebench 1.4.9.1. http://sourceforge.net/projects/filebench/, 2011.

[51] TANENBAUM, A. S. *Modern Operating Systems*, 3rd ed. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

[52] THERESKA, E., SCHINDLER, J., BUCY, J., SALMON, B., LUMB, C. R., AND GANGER, G. R. A framework for building unobtrusive disk maintenance applications. In *Proceedings of the USENIX Conference on File and Storage Technologies* (March 2004), FAST '04.

[53] USECHE, L., KOLLER, R., RANGASWAMI, R., AND VERMA, A. Truly non-blocking writes. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems* (June 2011), HotStorage '11.

[54] VENKATARAMAN, S., TOLIA, N., RANGANATHAN, P., AND CAMPBELL, R. H. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the USENIX Conference on File and Storage Technologies* (February 2011), FAST '11.

[55] VMWARE, INC. VMware Virtual SAN. http://www.vmware.com/products/virtual-san/, 2013.

[56] VOLOS, H., TACK, A. J., AND SWIFT, M. Mnemosyne: Lightweight persistent memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (2011), ASPLOS XVI.

[57] WU, X., AND REDDY, A. L. N. Exploiting concurrency to improve latency and throughput in a hybrid storage system. In *Proceedings of the IEEE International Symposium in Modeling, Analysis and Simulation of Computer and Telecommunication Systems* (September 2010), MASCOTS '10.

[58] ZHOU, P., ZHAO, B., YANG, J., AND ZHANG, Y. A durable and energy efficient main memory using pcm technology. In *Proceedings of the International Symposium on Computer Architecture* (2009), ISCA '09.

NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems

Jun Yang[1], Qingsong Wei[§ 1], Cheng Chen[1], Chundong Wang[1], Khai Leong Yong[1] and Bingsheng He[2]

[1]Data Storage Institute, A-STAR, Singapore
[2]Nanyang Technological University

Abstract

The non-volatile memory (NVM) has DRAM-like performance and disk-like persistency which make it possible to replace both disk and DRAM to build single level systems. To keep data consistency in such systems is non-trivial because memory writes may be reordered by CPU and memory controller. In this paper, we study the consistency cost for an important and common data structure, B^+Tree. Although the memory fence and CPU cacheline flush instructions can order memory writes to achieve data consistency, they introduce a significant overhead (more than 10X slower in performance). Based on our quantitative analysis of consistency cost, we propose NV-Tree, a consistent and cache-optimized B^+Tree variant with reduced CPU cacheline flush. We implement and evaluate NV-Tree and NV-Store, a key-value store based on NV-Tree, on an NVDIMM server. NV-Tree outperforms the state-of-art consistent tree structures by up to 12X under write-intensive workloads. NV-Store increases the throughput by up to 4.8X under YCSB workloads compared to Redis.

1 Introduction

For the past few decades, DRAM has been de facto building block for the main memory of computer systems. However, it is becoming insufficient with an increasing need of large main memory due to its density limitation [40, 43]. To address this issue, several Non-Volatile Memory (NVM) technologies have been under active development, such as phase-change memory (PCM) [49], and spin-transfer torque memory (STT-RAM) [29]. These new types of memory have the potential to provide comparable performance and much higher capacity than DRAM. More important, they are persistent which makes failure recovery faster [31, 33].

Considering the projected cost [21] and power efficiency of NVM, there have been a number of proposals that replace both disk and DRAM with NVM to build a single level system [21, 53, 45]. Such systems can (i) eliminate the data movement between disk and memory, (2) fully utilize the low-latency byte-addressable NVM by connecting it through memory bus instead of legacy block interface [16, 30, 56, 7, 6]. However, with data stored only in NVM, data structures and algorithms must be carefully designed to avoid any inconsistency caused by system failure. In particular, if the system crashes when an update is being made to a data structure in NVM, the data structure may be left in a corrupted state as the update is only half-done. In that case, we need certain mechanism to recover the data structure to its last consistent state. To achieve data consistency in NVM, ordered memory writes is fundamental. However, existing CPU and memory controller may reorder memory writes which makes it non-trivial to develop consistent NVM-based systems and data structures, as demonstrated in previous works [44, 53, 58, 55, 14, 12, 18, 35, 22, 46, 10, 32, 17]. To maintain memory writes to NVM in certain order, we must (1) prevent them from being reordered by CPU and (2) manually control CPU cacheline flush to make them persistent on NVM. Most studies use CPU instructions such as memory fence and cacheline flush. However, these operations introduce significant overhead [14, 53, 44]. We observe a huge amplification of CPU cacheline flush when using existing approaches to keep B^+Tree [13] consistent, which makes the consistency cost very high.

In this paper, we propose **NV-Tree**, a consistent and cache-optimized B^+Tree variant which reduces CPU cacheline flush for keeping data consistency in NVM. Specifically, NV-Tree decouples tree nodes into two parts, leaf nodes (LNs) as *critical data* and internal nodes (INs) as *reconstructable data*. By enforcing consistency only on LNs and reconstructing INs from LNs during failure recovery, the consistency cost for INs is eliminated but the data consistency of the entire NV-Tree is still guaranteed. Moreover, NV-Tree keeps entries in each LN *unsorted* which can reduce CPU cacheline flush

[§] Corresponding author: WEI_Qingsong@dsi.a-star.edu.sg by 82% to 96% for keeping LN consistent. Last but not least, to overcome the drawback of slowing down searches and deletions due to the write-optimized design in LN, NV-Tree adopts a pointer-less layout for INs to further increase the CPU cache efficiency.

Our contributions can be summarized as follows:

1. We quantify the consistency cost for B^+Tree using existing approaches, and present two insightful observations: (1) keeping entries in LN sorted introduces large amount of CPU cacheline flush which dominates the overall consistency cost (over 90%); (2) enforcing consistency only on LN is sufficient to keep the entire tree consistent because INs can always be reconstructed even after system failure.
2. Based on the observations, we present our **NV-Tree**, which (1) decouples LNs and INs, only enforces consistency on LNs; (2) keeps entries in LN unsorted, updates LN consistently without logging or versioning; (3) organizes INs in a cache-optimized format to further increase CPU cache efficiency.
3. To evaluate NV-Tree in system level, we have also implemented a key-value store, called **NV-Store**, using NV-Tree as the core data structure.
4. Both NV-Tree and NV-Store are implemented and evaluated on a real NVDIMM [1] platform. The experimental results show that NV-Tree outperforms CDDS-Tree [53], the state-of-art consistent tree structure, by up to **12X** under write-intensive workloads. The speedup drops but still reaches 2X under read-intensive workloads. NV-Store increases the throughput by up to **4.8X** under YCSB workloads compared to Redis [50].

The rest of this paper is organized as follows. Section 2 discusses the background, related work and motivation. Section 3 presents the detailed design and implementation of NV-Tree. The experimental evaluation of NV-Tree and NV-Store is shown in Section 4. Finally, Section 5 concludes this paper.

2 Related Work and Motivation

2.1 Non-Volatile Memory (NVM)

Computer memory has been evolving rapidly in recent years. A new category of memory, NVM, has attracted more and more attention in both academia and industry [21, 55]. Early work [36, 41, 35, 57, 47, 51, 11, 52, 59, 60, 2, 27, 39] focuses on flash memory. As shown in Table 1, flash is faster than HDD but is still unsuitable to replace DRAM due to much higher latency and limited endurance [24]. Recent work has focused on the next generation NVM [28], such as PCM [49, 42, 4, 8, 23] and STT-RAM [29], which (i) is byte addressable, (ii) has DRAM-like performance, and (iii) provides better endurance than flash. PCM is several times slower than DRAM and its write endurance is limited to as few as 10^8 times. However, PCM has larger density than DRAM and shows a promising potential for increasing the capacity of main memory. Although wear-leveling is necessary for PCM, it can be done by memory controller [48, 61]. STT-RAM has the advantages of lower power consumption over DRAM, unlimited write cycles over PCM, and lower read/write latency than PCM. Recently, Everspin announced its commercial 64Mb STT-RAM chip with DDR3 interface [20]. In this paper, **NVM** is referred to the next generation of non-volatile memory excluding flash memory.

Table 1: Characteristics of Different Types of Memory

Category	Read Latency (*ns*)	Write Latency (*ns*)	Endurance (*# of writes per bit*)
SRAM	2-3	2-3	∞
DRAM	15	15	10^{18}
STT-RAM	5-30	10-100	10^{15}
PCM	50-70	150-220	10^8-10^{12}
Flash	25,000	200,000-500,000	10^5
HDD	3,000,000	3,000,000	∞

Due to the price and prematurity of NVM, mass production with large capacity is still impractical today. As an alternative, NVDIMM [44], which is commercially available [1], provides persistency and DRAM-like performance. NVDIMM is a combination of DRAM and NAND flash. During normal operations, NVDIMM is working as DRAM while flash is invisible to the host. However, upon power failure, NVDIMM saves all the data from DRAM to flash by using supercapacitor to make the data persistent. Since this process is transparent to other parts of the system, NVDIMM can be treated as NVM. In this paper, our NV-Tree and NV-Store are implemented and evaluated on a NVDIMM platform.

2.2 Data Consistency in NVM

NVM-based single level systems [21, 14, 53, 58, 44] have been proposed and evaluated using the simulated NVM in terms of cost, power efficiency and performance. As one of the most crucial features of storage systems, data consistency guarantees that stored data can survive system failure. Based on the fact that data is recognizable only if it is organized in a certain format, updating data consistently means preventing data from being lost or partially updated after a system failure. However, the atomicity of memory writes can only be supported with a very small granularity or no more than the memory bus width (8 bytes for 64-bit CPUs) [25] which is addressed in previous work [55, 14, 44], so updating data larger than 8 bytes requires certain mechanisms to make sure data can be recovered even if system failure happens before it is completely updated. Particularly, the approaches such as logging and copy-on-write make data recoverable by writing a copy elsewhere before updating the data itself. To implement these approaches, we must make sure memory writes are in a certain order, e.g., the memory writes for making the copy of data must be completed before updating the data itself. Similar write ordering requirement also exists in pointer-based data structures, e.g., in B^+Tree, if one tree node is split, the new node must be written completely before its pointer being added to the parent node, otherwise, the wrong write order will make the parent node contain an invalid pointer if the system crash right after the pointer being added.

Unfortunately, memory writes may be reordered by either CPU or memory controller. Alternatively, without modifying existing hardware, we can use the sequence of {`MFENCE`, `CLFLUSH`, `MFENCE`} instruction (referred to `flush` in the rest of this paper) to form ordered memory writes [53]. Specifically, `MFENCE` issues a memory barrier which guarantees the memory operations after the barrier cannot proceed until those before the barrier complete, but it does not guarantee the order of write-back to the memory from CPU cache. On the other hand, `CLFLUSH` can explicitly invalidate the corresponding dirty CPU cachelines so that they can be flushed to NVM by CPU which makes the memory write persistent eventually. However, `CLFLUSH` can only flush a dirty cacheline by explicitly invalidating it which makes CPU cache very inefficient. Although such invalidations can be avoided if we can modify the hardware itself to implement *epoch* [14], CPU cacheline flush cannot be avoided. Reducing it is still necessary to not only improve performance but also extend the life cycle of NVM with reduced memory write.

2.3 Related Work

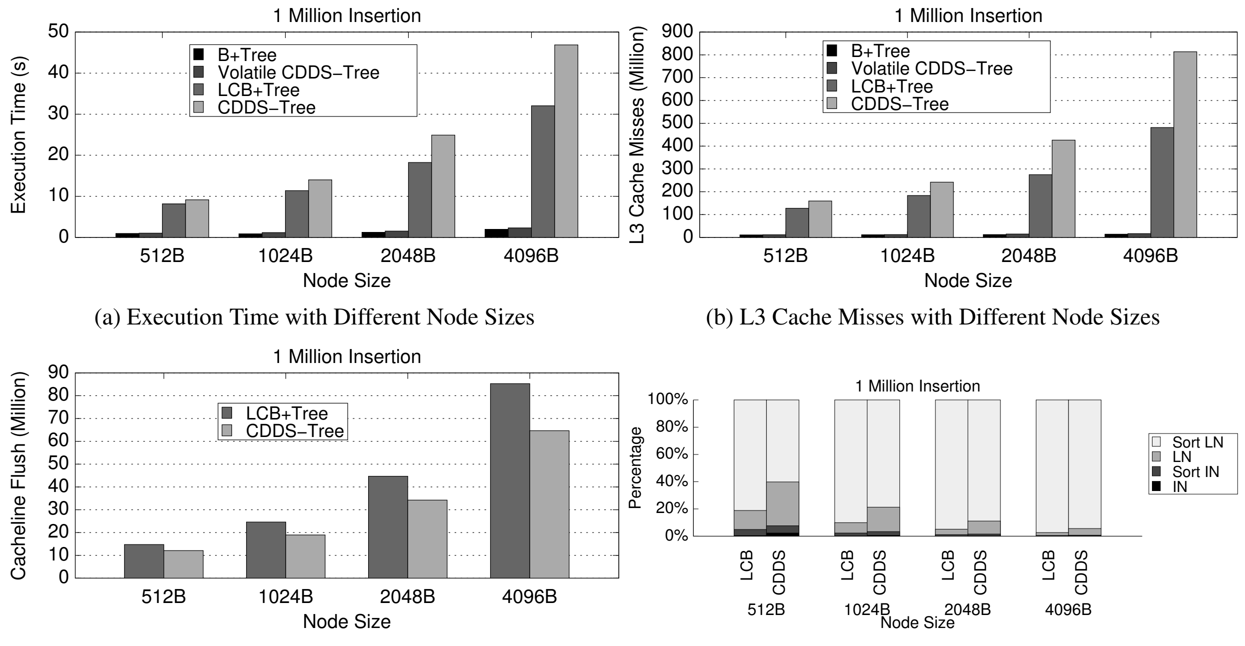
Recent work proposed mechanisms to provide data consistency in NVM-based systems by either modifying existing hardware or using CPU primitive instructions such as `MFENCE` and `CLFLUSH`. BPFS [14] proposed a new file system which is resided in NVM. It adopts a copy-on-write approach called short-circuit shadow paging using *epoch* which can flush dirty CPU cachelines without invalidating them to order memory writes for keeping data consistency. However, it still suffers from the overhead of cacheline flush. It must be implemented by modifying existing hardware which is not practical in most cases. Volos et al. [55] proposed Mnemosyne, a new program interface for memory allocations in NVM. To manage memory consistency, it presents persist memory region, persist primitives and durable memory transaction which consist of `MFENCE` and `CLFLUSH` eventually. NV-Heaps [12] is another way to consistently manage NVM directly by programmers based on *epoch*. It uses *mmap* to access spaces in NVM and gives a way to allocate, use and deallocate objects and their pointers in NVM. Narayanan et al. [44] proposed a way to keep the whole system status when power failure happens. Realizing the significant overhead of flushing CPU cacheline to NVM, they propose to flush-on-fail instead of flush-on-demand. However, they cannot protect the system from any software failure. In general, flushing CPU cacheline is necessary to order memory writes and used in almost all the existing NVM-based systems [34, 37, 54, 19].

The most related work to our NV-Tree is CDDS-Tree [53] which uses `flush` to enforce consistency on all the tree nodes. In order to keep entries sorted, when an entry is inserted to a node, all the entries on the right side of the insertion position need to be shifted. CDDS-Tree performs `flush` for each entry shift, which makes the consistency cost very high. Moreover, it uses the entry-level versioning approach to keep consistency for all tree operations. Therefore, a background garbage collector and a relatively complicated recovery process are both needed.

2.4 Motivation

To quantify the consistency cost, we compare the execution of performing one million insertion in (a) a standard B^+Tree [13] without consistency guarantee, (b) a log-based consistent B^+Tree (**LCB$^+$Tree**), (c) a CDDS-Tree [53] using versioning, and (d) a volatile CDDS-Tree with `flush` disabled. In LCB$^+$Tree, before modifying a node, its original copy is logged and `flush`ed. The modified part of it is then `flush`ed to make the changes persistent. Note that we only use LCB$^+$Tree as the baseline to illustrate one way to use logging to guarantee the consistency. We understand optimizations (such as combining several modification to one node into one flush) can be made to improve the performance of LCB$^+$Tree but it is beyond the scope of this paper. Since CDDS-Tree is not open-source, we implement it ourselves and achieve similar performance to that in the original paper [53]. As shown in Figure 1a, for one million insertion with 4KB nodes, the LCB$^+$Tree and CDDS-Tree are up to 16X and 20X slower than their volatile version, respectively. Such performance drop is caused by the increased number of cache misses and additional cacheline flush.

Remembering that `CLFLUSH` flushes a dirty cacheline by explicitly invalidating it, which causes a cache miss



(a) Execution Time with Different Node Sizes (b) L3 Cache Misses with Different Node Sizes

(c) Total Number of CPU Cacheline Flush (d) Percentage Breakdown of CPU Cacheline Flush

Figure 1: Consistency Cost Analysis of B^+Tree and CDDS-Tree

when reading the same memory address later. We use Intel vTune Amplifier[1], a CPU profiling tool, to count the L3 cache misses during the one million insertion. As shown in Figure 1b, while the volatile CDDS-Tree or B^+Tree produces about 10 million L3 cache misses, their consistent version causes about 120-800 million cache misses which explains the performance drop.

Figure 1c shows the total number of cacheline flushes in CDDS-Tree and LCB^+Tree for one million insertion. With 0.5KB/1KB/2KB/4KB nodes, the total amount of cacheline flushes is 14.8/24.6/44.7/85.26 million for LCB^+Tree, and 12.1/19.0/34.2/64.7 million for CDDS-Tree. This indicates that **keeping consistency causes a huge amplification of the CPU cacheline invalidation and flush, which increases the cache misses significantly**, as shown in Figure 1b.

The numbers of both the cache misses and cacheline flushes in LCB^+Tree and CDDS-Tree are proportional to the node size due to the `flush` for keeping the entries sorted. Specifically, for LCB^+Tree and CDDS-Tree, all the shifted entries caused by inserting an entry inside a node need to be `flush`ed to make the insertion persistent. As a result, the amount of data to be `flush`ed is related to the node size for both trees.

We further categorize the CPU cacheline flush into four types, as shown in Figure 1d, *Sort LN*/*Sort IN* stands for the cacheline flush of shifted entries. It also includes the `flush` of logs in LCB^+Tree. *LN*/*IN* stands for the flush of other purpose such as `flush`ing new nodes and updated pointers after split, etc. The result shows that **the consistency cost due to `flush` mostly comes from `flush`ing shifted entries in order to keep LN sorted**, about 60%-94% in CDDS-Tree, and 81%-97% in LCB^+Tree.

Note that CDDS-Tree is slower than LCB^+Tree by 11-32% even though it produces less cacheline flush. The reasons are that (1) the size of each `flush` in CDDS-Tree is the entry size, which is much smaller than that in LCB^+Tree, and (2) the performance of `flush` for small objects is over 25% slower than that for large objects [53].

Last but not least, we observe that given a data structure, **not all the data needs to be consistent to keep the entire data structure consistent.** As long as some parts of it (denoted as *critical data*) is consistent, the rest (denoted as *reconstructable data*) can be reconstructed without losing consistency for the whole data structure. For instance, in B^+Tree, where all the data is stored in LNs, they can be considered as *critical data* while INs are *reconstructable data* because they can always be reconstructed from LNs at a reasonably low cost. That suggests we may only need to enforce consistency on *critical data*, and reconstruct the entire data structure from the consistent *critical data* during the recovery.

[1]https://software.intel.com/en-us/intel-vtune-amplifier-xe

3 NV-Tree Design and Implementation

In this section, we present NV-Tree, a consistent and cache-optimized B$^+$Tree variant with reduced consistency cost.

3.1 Design Decisions

Based on our observations above, we make three major design decisions in our NV-Tree as the following.

$\mathcal{D}1$. ***Selectively Enforce Data Consistency.*** NV-Tree decouples LNs and INs by treating them as *critical data* and *reconstructable data*, respectively. Different from the traditional design where all nodes are updated with consistency guaranteed, NV-Tree only enforces consistency on LNs (*critical data*) but processes INs (*reconstructable data*) with no consistency guaranteed to reduce the consistency cost. Upon system failure, INs are reconstructed from the consistent LNs so that the whole NV-Tree is always consistent.

$\mathcal{D}2$. ***Keep Entries in LN Unsorted.*** NV-Tree uses unsorted LNs so that the `flush` operation used in LCB$^+$Tree and CDDS-Tree for shifting entries upon insertion can be avoided. Meanwhile, entries of INs are still sorted to optimize search performance. Although the unsorted LN strategy is not new [9], we are the first one that quantify its impact on the consistency cost and propose to use it to reduce the consistency cost in NVM. Moreover, based on our unsorted scheme for LNs, both the content (entry insertion/update/deletion) and structural (split) changes in LNs are designed to be visible only after a CPU primitive atomic write. Therefore, LNs can be protected from being corrupted by any half-done updates due to system failure without using logging or versioning. Thanks to the invisibility of on-going updates, the parallelism of accessing LN is also increased because searching in LN is no longer blocked by the concurrent on-going update.

$\mathcal{D}3$. ***Organizing IN in Cache-optimized Format.*** The CPU cache efficiency is a key factor to the performance. In NV-Tree, all INs are stored in a consecutive memory space and located by offset instead of pointers, and all nodes are aligned to CPU cacheline. As a result, NV-Tree achieves higher space utilization and cache hit rate.

3.2 NV-Tree

In this subsection, we present the details of tree node layout design and all the tree operations of NV-Tree.

3.2.1 Overview

In NV-Tree, as shown in Figure 2, all the data is stored in LNs which are linked together with right-sibling pointers. Each LN can also be accessed by the LN pointer stored in the last level of IN, denoted as PLN (parent of

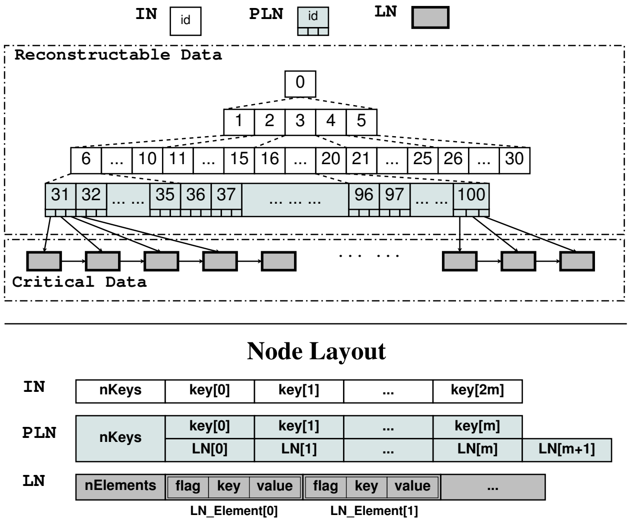


Figure 2: NV-Tree Overview and Node Layout

leaf node). All the IN/PLNs are stored in a pre-allocated consecutive memory space which means the position of each IN/PLN is fixed upon creation. The *node id* of each IN/PLN is assigned sequentially from 0 (root). Therefore it can be used to calculate the offset of each IN/PLN to the root. Given the memory address of the root, all the IN/PLNs can be located without using any pointers. Each key/value pair (KV-pair) stored in LNs is encapsulated in an *LN_element*.

Keeping each LN and the LN list consistent in NV-Tree without using logging or versioning is non-trivial. Different from a normal B$^+$Tree, both update and deletion are implemented as insertion using an append-only strategy discussed in Section 3.2.3. Any insertion/update/deletion operations may lead to a full LN which triggers either *split/replace/merge* discussed in Section 3.2.4. We carefully design the write order for insertion (update/deletion) and split/replace/merge using `flush` to guarantee the changes made by these operations cannot be seen until a successful atomic write. When one PLN is full, a procedure called *rebuilding* is executed to reconstruct a new set of IN/PLN to accommodate more LNs, discussed in Section 3.2.5.

3.2.2 Locating Target LN

We first present how to find the target LN in NV-Tree. Due to the hybrid design, the procedure of locating target LN with a given key in NV-Tree is different from that in standard B$^+$Tree.

As shown in Algorithm 1, given the search key and the memory address of root, INs are searched level by level, starting from root with node id 0. On each level, which child to go in the next level is determined by a binary search based on the given search key. For instance, with

Algorithm 1: NV-Tree LN Lookup

```
1 Function find_leaf(k, r)
      Input: k: key, r: root
      Output: LNpointer: the pointer of target leaf
              node
      /* Start from root (id=0).     */
2     $id \leftarrow 0$;
3     while $id \notin PLNIDs$ do /* Find PLN.     */
4         $IN \leftarrow$ memory address of node id;
5         $pos \leftarrow BinarySearch(key, IN)$;
6         $id \leftarrow id * (2m+1) + 1 + pos$;
          /* m is the maximum number of
             keys in a PLN.           */
7     $PLN \leftarrow$ memory address of node id;
8     $pos \leftarrow BinarySearch(key, PLN)$;
9     return PLN.LNpointers[pos]
```

keys and pointers having the same length, if a PLN can hold m keys and $m+1$ LN pointers, an IN can hold $2m$ keys. If the node id of current IN is i and the binary search finds the smallest key which is no smaller than the search key is at position k in current IN, then the next node to visit should have the node id $(i \times (2m+1) + 1 + k)$. When reaching a PLN, the address of the target LN can be retrieved from the leaf node pointer array.

As every IN/PLN has a fixed location once rebuilt, PLNs are not allowed to split. Therefore, the content of INs (PLNs excluded) remains unchanged during normal execution. Therefore, NV-Tree does not need to use locks in INs for concurrent tree operations which increases the scalability of NV-Tree.

3.2.3 Insertion, Update, Deletion and Search

Insertion starts with finding target LN. After target LN is located, a new *LN_element* will be generated using the new KV-pair. If the target LN has enough space to hold the *LN_element*, the insertion completes after the *LN_element* is **appended**, and the *nElement* is increased by one successfully. Otherwise, the target LN will split before insertion (discussed in Section 3.2.4). The pseudo-code of insertion is shown in Algorithm 2. Figure 3a shows an example of inserting a KV-pair `{7,b}` into an LN with existing two KV-pairs `{6,a}` and `{8,c}`.

Deletion is implemented just the same as insertion except a special NEGATIVE flag. Figure 3b shows an example of deleting the `{6,a}` in the original LN. A NEGATIVE *LN_element* `{6,a}` (marked as '-') is inserted. Note that the NEGATIVE one cannot be inserted unless a normal one is found. The space of both the NEGATIVE and normal *LN_element*s are recycled by later split.

Algorithm 2: NV-Tree Insertion

```
   Input: k: key, v: value, r: root
   Output: SUCCESS/FAILURE
1  begin
2      if $r = NULL$ then /* Create new tree
       with the given KV-pair.        */
3          $r \leftarrow$ create_new_tree(k, v);
4          return SUCCESS
5      $leaf \leftarrow$ find_leaf(k, r);
6      if LN has space for new KV-pair then
7          $newElement \leftarrow CreateElement(k, v)$;
8          flush(newElement);
9          $AtomicInc(leaf.number)$;
10         flush(leaf.number);
11     else
12         leaf_split_and_insert(leaf, k, v)
13     return SUCCESS
```

Update is implemented by inserting two *LN_element*s, a NEGATIVE with the same *value* and a normal one with updated *value*. For instance, as shown in Figure 3c, to update the original `{8,c}` with `{8,y}`, the NEGATIVE *LN_element* for `{8,c}` and the normal one for `{8,y}` are appended accordingly.

Note that the order of appending *LN_element* before updating *nElement* in LN is guaranteed by `flush`. The appended *LN_element* is only visible after the *nElement* is increased by a successful atomic write to make sure LN cannot be corrupted by system failure.

Search a key starts with locating the target LN with the given key. After the target LN is located, since keys are unsorted in LN, a scan is performed to retrieve the *LN_element* with the given *key*. Note that if two *LN_element*s have the target key and same *value* but one of them has a NEGATIVE flag, both of them are ignored because that indicates the corresponding KV-pair is deleted. Although the unsorted leaf increases the searching time inside LN, the entries in IN/PLNs are still sorted so that the search performance is still acceptable as shown in Section 4.4.

All the modification made to LNs/PLNs is protected by light-weight latches. Meanwhile, given the nature of the append-only strategy, searching in LNs/PLNs can be executed without being blocked by any ongoing modification as long as the *nElement* is used as the boundary of the search range in LNs/PLNs.

3.2.4 LN Split/Replace/Merge

When an LN is full, the first step is to scan the LN to identify the number of valid *LN_element*s. Those NEGATIVE ones and the corresponding normal ones are

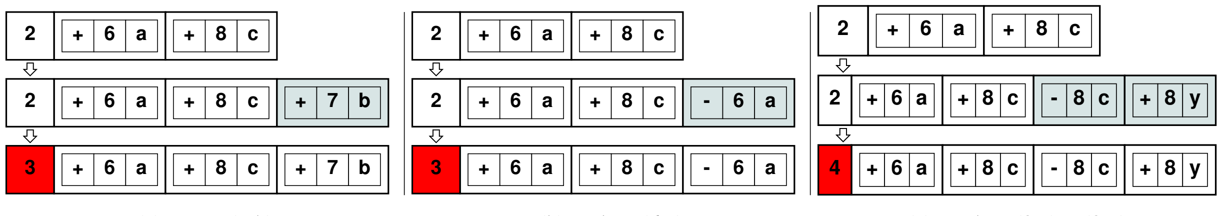
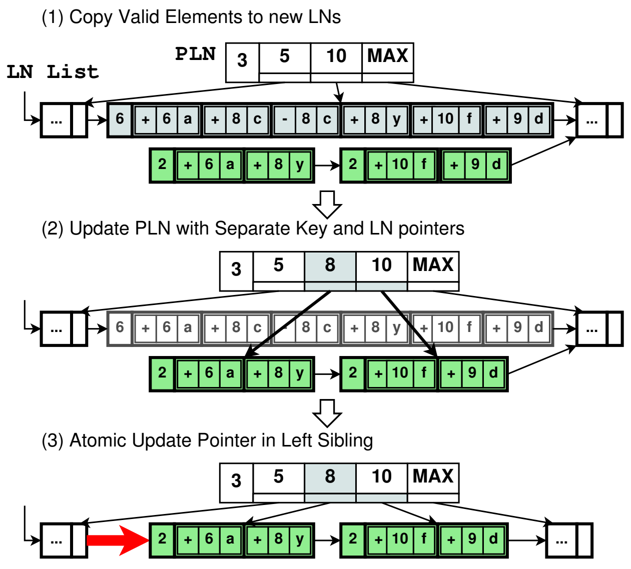


Figure 3: Example of NV-Tree Insertion/Deletion/Update


(1) Copy Valid Elements to new LNs
LN List
PLN
(2) Update PLN with Separate Key and LN pointers
(3) Atomic Update Pointer in Left Sibling
Figure 4: Example of LN Split

both considered invalid. The second step is to determine whether the LN needs a split.

If the percentage of valid elements is above the minimal fill factor (e.g., 50% in standard B$^+$Tree), we perform **split**. Two new LNs (left and right) are created and valid elements are copied to either of them according to the selected separate key. Then the new KV-pair is inserted accordingly. The split completes after the pointer in the left sibling of the old LN is updated to point to new left LN using an atomic write. Before that, all the changes made during split are not visible to the tree. Figure 4 shows an example of an LN split.

If the percentage is below the minimal fill factor, we check the number of *LN_element*s in the right sibling of the old LN. If it is above the minimal fill factor, we perform **replace**, otherwise, we perform **merge**. For **replace**, those valid *LN_element*s in the old LN are copied to a new LN, and the new LN replaces the old LN in the LN list using an atomic write. For **merge**, those valid *LN_element*s from both the old LN and its right sibling are copied to a new LN, and the new LN replaces both of them in the LN list using an atomic write. Note that we use the *nElement* instead of the number of valid elements in the right sibling to decide which operation to perform because finding the latter needs to perform a scan which is relatively more expensive. Due to space limitation, examples of *replace* and *merge* are omitted here.

3.2.5 Rebuilding

As the memory address of each IN/PLN is fixed upon creation, IN/PLNs are not allowed to split. Therefore, when one PLN is full, all IN/PLNs have to be reconstructed to make space in PLNs to hold more LN pointers. The first step is to determine the new number of PLNs based on the current number of LNs. In our current implementation, to delay the next rebuilding as much as possible under a workload with uniformly distributed access pattern, each PLN stores exactly one LN pointer after rebuilding. Optimizing rebuilding for workloads with different access patterns is one of our future work.

During normal execution, we can use *rebuild-from-PLN* strategy by redistributing all the keys and LN pointers in existing PLNs into the new set of PLNs. However, upon system failure, we use *rebuild-from-LN* strategy. Because entries are unsorted in each LN, *rebuild-from-LN* needs to scan each LN to find its maximum key to construct the corresponding key and LN pointer in PLN. *Rebuild-from-LN* is more expensive than *rebuild-from-PLN* but is only executed upon system failure. Compared to a single tree operation (e.g., insertion or search), one rebuilding may be very time-consuming in large NV-Tree. However, given the frequency of rebuilding, such overhead is neglectable in a long-running application (less than 1% in most cases, details can be found in Section 4.7).

If the memory space is enough to hold the new IN/PLNs without deleting the old ones, search can still proceed during rebuilding because it can always access the tree from the old IN/PLNs. In that case, the memory requirement of rebuilding is the total size of both old and new IN/PLNs. For instance, when inserting 100 million entries with random keys to a NV-Tree with 4KB nodes, rebuilding is executed only for two times. The memory requirement to enable parallel rebuilding for the first/second rebuilding is only about 1MB/129MB which is totally acceptable.

3.2.6 Recovery

Since the LN list (*critical data*) is consistent, *rebuild-from-LN* is sufficient to recover a NV-Tree from either normal shutdown or system failure.

To further optimize the recovery after normal shutdown, our current implementation is able to achieve **instant recovery** by storing IN/PLNs persistently in NVM. More specifically, during normal shutdown, we (1) `flush` all IN/PLNs to NVM, (2) save the root pointer to a reserved position in NVM, (3) and use an atomic write to mark a special flag along with the root pointer to indicate a successful shutdown. Then, the recovery can (1) start with checking the special flag, (2) if it is marked, reset it and use the root pointer stored in NVM as the current root to complete the recovery. Otherwise, it means a system failure occurred, and a *rebuild-from-LN* procedure is executed to recover the NV-Tree.

4 Evaluation

In this section, we evaluate our NV-Tree by comparing it with LCB$^+$Tree and CDDS-Tree in terms of insertion performance, overall performance under mixed workloads and throughput of all types of tree operations. We also study the overhead of rebuilding by quantifying its impact on the overall performance. We use YCSB [15], a benchmark for KV-stores, to perform an end-to-end comparison between our NV-Store and Redis [50], a well-known in-memory KV-store. Finally, we discuss the performance of NV-Tree on different types of NVM and estimated performance with *epoch*.

4.1 Implementation Effort

We implement our NV-Tree from scratch, an LCB$^+$Tree by applying `flush` and logging to a standard B$^+$Tree [5], and a CDDS-Tree [53]. To make use of NVDIMM as a persistent storage device, we modify the memory management of Linux kernel to add new functions (e.g., `malloc_NVDIMM`) to directly allocate memory space from NVDIMM. The NVDIMM space used by NV-Tree is guaranteed to be mapped to a continuous (virtual) memory space. The node "pointer" stored in NV-Tree is actually the memory offset to the start address of the mapped memory space. Therefore, even if the mapping is changed after reboot, each node can always be located using the offset. With the position information of the first LN stored in a reserved location, our NV-Tree is practically recoverable after power down.

We build our NV-Store based on NV-Tree by allowing different sizes of key and value. Moreover, by adding a timestamp in each *LN_Element*, NV-Store is able to support lock-free concurrent access using timestamp-based multi-version concurrency control (MVCC) [38]. Based on that, we implement NV-Store to support Snapshot Isolation [3] transactions. Finally, we implement a database interface layer to extend YCSB to support NV-Store to facilitate our performance evaluation.

4.2 Experimental Setup

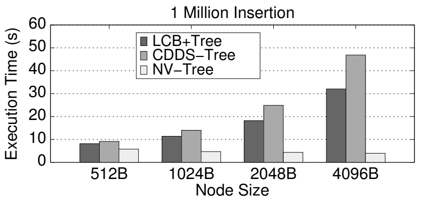
All of our experiments are conducted on a Linux server (Kernel version 3.13.0-24) with an Intel Xeon E5-2650 2.4GHz CPU (512KB/2MB/20MB L1/L2/L3 cache), 8GB DRAM and 8GB NVDIMM [1] which has practically the same read/write latency as DRAM. In the end-to-end comparison, we use YCSB (0.1.4) to compare NV-Store with Redis (2.8.13). Note that all results shown in this section are produced by running application on NVDIMM server instead of simulation. The execution time measured for NV-Tree and NV-Store includes the rebuilding overhead.

4.3 Insertion Performance

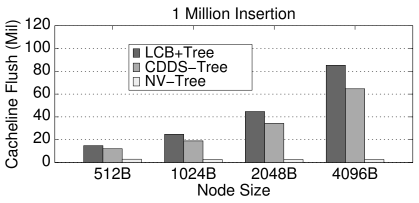
We first compare the insertion performance of LCB$^+$Tree, CDDS-Tree and NV-Tree with different node sizes. Figure 5a shows the execution time of inserting one million KV-pairs (8B/8B) with randomly selected keys to each tree with different sizes of tree nodes from 512B to 4KB. The result shows that NV-Tree outperforms LCB$^+$Tree and CDDS-Tree up to 8X and 16X with 4KB nodes, respectively. Moreover, different from LCB$^+$Tree and CDDS-Tree that the insertion performance drops when the node size increases, NV-Tree shows the best performance with larger nodes. This is because (1) NV-Tree adopts unsorted LN to avoid CPU cacheline flush for shifting entries. The size of those cacheline flush is proportional to the node size in LCB$^+$Tree and CDDS-Tree; (2) larger nodes lead to less LN split resulting in less rebuilding and reduced height of NV-Tree.

The performance improvement of NV-Tree over the competitors is mainly because of the reduced number of cacheline flush thanks to both the unsorted LN and decoupling strategy of enforcing consistency selectively. Specifically, as shown in Figure 5b, NV-Tree reduces the total CPU cacheline flush by 80%-97% compared to LCB$^+$Tree and 76%-96% compared to CDDS-Tree.

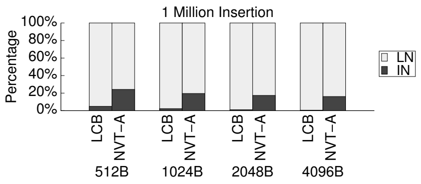
Although the consistency cost of INs is almost neglectable for LCB$^+$Tree and CDDS-Tree as shown in Figure 1d, such cost becomes relatively expensive in NV-Tree. This is because the consistency cost for LN is significantly reduced after our optimization for LN, such as keeping entries unsorted and modifying LN with a log-free append-only approach. To quantify the consistency cost of INs after such optimization, we implement a mod-



(a) Execution Time with Varied Node Sizes



(b) Number of CPU Cacheline Flush



(c) Cacheline Flush Breakdown for IN/LN

Figure 5: Insertion Performance and Cacheline Flush Comparison

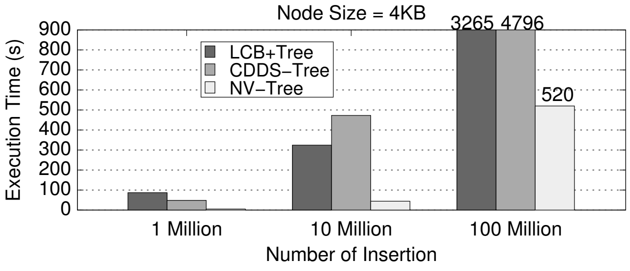


Figure 6: Execution Time of 1/10/100 Million Insertion

ified NV-Tree, denoted as NVT-A, which does the same optimization for LN as NV-Tree, but manages INs in the same way as LCB$^+$Tree and enforces consistency for all INs. Figure 5c shows the breakdown of CPU cacheline flush for IN and LN in LCB$^+$Tree and NVT-A. The percentage of CPU cacheline flush for IN increase from around 7% in LCB$^+$Tree to more than 20% in NVT-A. This result proves that decoupling IN/LN and enforcing consistency selectively are necessary and beneficial.

Figure 6 shows the execution time of inserting different number of KV-pairs with 4KB node size. The result shows that for inserting 1/10/100 million KV-pairs, the speedup of NV-Tree can be 15.2X/6.3X/5.3X over LCB$^+$Tree and 8X/9.7X/8.2X over CDDS-Tree. This suggests that although inserting more KV-pairs increases the number and duration of rebuilding, NV-Tree can still outperform the competitors thanks to the write-optimized design.

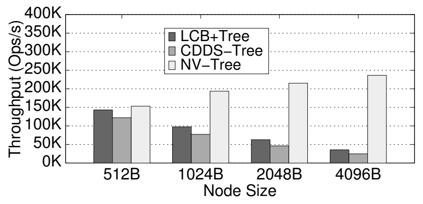
4.4 Update/Deletion/Search Throughput

This subsection compares the throughput of update/deletion/search operations in LCB$^+$Tree, CDDS-Tree and NV-Tree. In this experiment, we first insert one million KV-pairs, then update each of them with new value (same size), then search with every key and finally delete all of them. For each type of operation, each key is randomly and uniquely selected. After each type of operation, we flush the CPU cache to remove the cache influence between different types of operation.

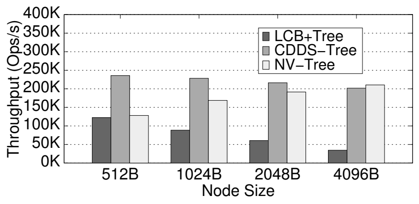
The update/deletion/search performance with node size varied from 512B to 4KB is shown in Figure 7. As shown in Figure 7a, NV-Tree improves the throughput of update by up to 5.6X and 8.5X over LCB$^+$Tree and CDDS-Tree. In CDDS-Tree, although update does not trigger the split if any reusable slots are available, entry shifting is still needed to keep the entries sorted. LCB$^+$Tree does not need to shift entries for update, but in addition to the updated part of the node, it `flush`es the log which contains the original copy of the node. In contrast, NV-Tree uses log-free append-only approach to modify LNs so that only two *LN_element*s need to be `flushed`.

Upon deletion, NV-Tree is better than LCB$^+$Tree but not as good as CDDS-Tree as shown in 7b. This is because CDDS-Tree simply does an in-place update to update the end version of a corresponding key. However, with the node size increased, NV-Tree is able to achieve comparable throughput to CDDS-Tree because of the reduction of split.

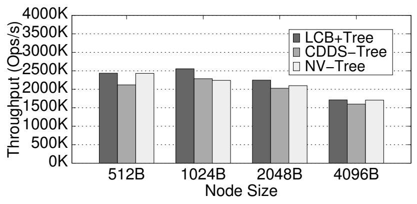
Note that the throughput of update and deletion in Figure 7a and 7b in LCB$^+$Tree decreases when the node size increases. This is because both the log size and the amount of data to `flush` for shifting entries are proportional to the node size. The same trend is observed in CDDS-Tree. In NV-Tree, by contrast, the throughput of update and deletion always increases when the node size increases because (1) the amount of data to `flush` is irrelevant to the node size, (2) a larger node reduces the



(a) Update



(b) Deletion



(c) Search

Figure 7: Update/Deletion/Search Throughput Comparison

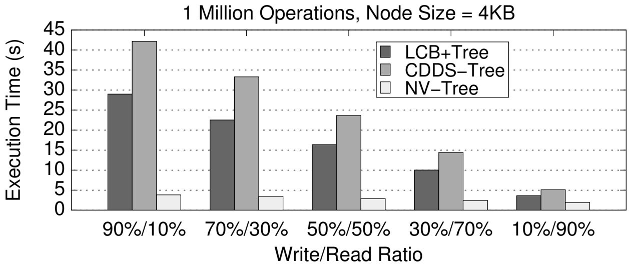


Figure 8: Execution Time of Mixed Workloads

number of split as well as rebuilding.

Although NV-Tree uses unsorted LN, thanks to the cache-optimized IN layout, the search throughput of NV-Tree is comparable to that of LCB$^+$Tree and CDDS-Tree as shown in Figure 7c, which is consistent to the published result [9].

4.5 Mixed Workloads

Figure 8 shows the execution time of performing one million insertion/search operations with varied ratios on an existing tree with one million KV-pairs.NV-Tree has the best performance under mixed workloads compared to LCB$^+$Tree and CDDS-Tree.

Firstly, all three trees have better performance under workloads with less insertion. This is because memory writes must be performed to write LN changes to NVM persistently through `flush` while searches can be much faster if they hit the CPU cache. Moreover, NV-Tree shows the highest speedup, 6.6X over LCB$^+$Tree and 10X over CDDS-Tree, under the most write-intensive workload (90% insertion/10% search). As the write/read ratio decreases, the speedup of NV-Tree drops but is still better than both competitors under the most read-intensive workload (10% insertion/90% search). This is because NV-Tree has much better insertion performance and comparable search throughput as well.

4.6 CPU Cache Efficiency

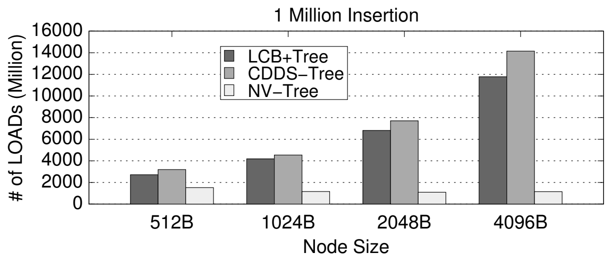
This subsection shows the underlying CPU cache efficiency of LCB$^+$Tree, CDDS-Tree and NV-Tree by using vTune Amplifier. Figure 9a shows the total number of LOAD instructions executed for inserting one million KV-pairs in each tree. NV-Tree reduces the number of LOAD instruction by about 44%-90% and 52%-92% compared to LCB$^+$Tree and CDDS-Tree, respectively. We also notice the number of LOAD instructions is not sensitive to the node size in NV-Tree while it is proportional to the node size in LCB$^+$Tree and CDDS-Tree. This is because NV-Tree (1) eliminates entry shifting during insertion in unsorted LN, (2) adopts cache-optimized layout for IN/PLNs.

Most important, NV-Tree produces much less cache misses. Since memory read is only needed upon L3 cache miss, we use the number of L3 cache misses to quantify the read penalty of `flush`. Figure 9b shows the total number of L3 cache misses when inserting one million KV-pairs. NV-Tree can reduce the number of L3 cache misses by 24%-83% and 39%-90% compared to LCB$^+$Tree and CDDS-Tree, respectively. This is because NV-Tree reduces the number of CPU cacheline invalidation and flush.

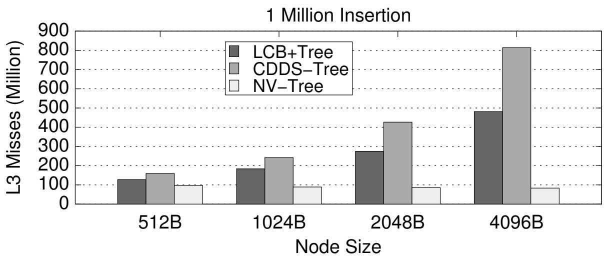
4.7 Rebuilding and Failure Recovery

To quantify the impact of rebuilding on the overall performance of NV-Tree, we measure the total number and time of rebuilding with different node sizes under different number of insertion. Compared to the total execution time, as shown in Table 2, the percentage of rebuilding time in the total execution time is below 1% for all types of workloads, which is totally neglectable. Moreover, we can tune the rebuilding frequency by increasing the size of tree nodes because the total number of splits decreases with larger nodes as shown in Figure 10a. With less splits, the frequency of rebuilding also becomes less, e.g., for 100 million insertion, with node size equals to 512B/1KB/2KB/4KB, the number of rebuilding is 7/4/3/2.

We also compare the performance of *rebuild-from-PLN* and *rebuild-from-LN*. Note that *rebuild-from-LN* is only used upon system failure. Figure 10b shows the total rebuilding time of both strategies for inserting 100 million KV-pairs to NV-Tree with different node sizes.



(a) Number of LOAD Instruction Executed

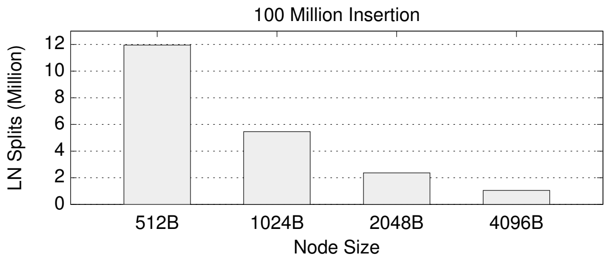


(b) Number of L3 Cache Miss

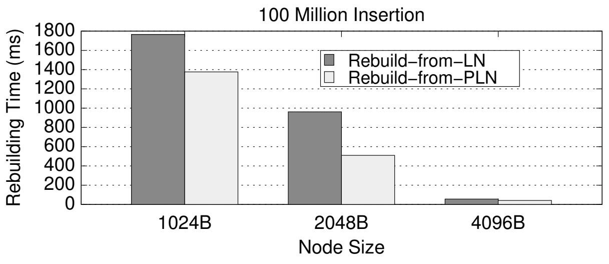
Figure 9: Cache Efficiency Comparison

Table 2: Rebuilding Time (***ms***) for 1/10/100 Million Insertion with 512B/1KB/2KB/4KB Nodes

Rebuild #	1M				10M				100M			
	Node Size				Node Size				Node Size			
	0.5KB	1KB	2KB	4KB	0.5KB	1KB	2KB	4KB	0.5KB	1KB	2KB	4KB
1	0.104	0.119	0.215	0.599	0.058	0.091	0.213	0.603	0.066	0.091	0.206	0.572
2	0.779	2.592	8.761	-	0.503	2.525	8.526	41.104	0.520	2.118	8.594	41.077
3	7.433	50.021	-	-	4.782	54.510	-	-	4.706	47.219	814.989	-
4	31.702	-	-	-	39.546	-	-	-	37.481	1310.004	-	-
5	-	-	-	-	312.139	-	-	-	322.606	-	-	-
6	-	-	-	-	-	-	-	-	2567.219	-	-	-
7	-	-	-	-	-	-	-	-	16231.647	-	-	-
Rebuilding Time	40.018	52.559	8.976	0.599	357.016	57.126	8.739	41.707	19164.135	1359.432	823.789	41.649
Execution Time	6107.971	4672.032	4349.421	3955.227	62649.634	55998.473	46874.810	44091.494	692341.866	604111.327	570825.594	518323.920
Percentage	**0.66%**	**1.13%**	**0.21%**	**0.02%**	**0.57%**	**0.10%**	**0.02%**	**0.09%**	**2.77%**	**0.23%**	**0.14%**	**0.01%**



(a) Number of LN Splits with Different Node Sizes



(b) Rebuilding Time for Different Rebuilding Strategy

Figure 10: Rebuilding Overhead Analysis

Rebuild-from-PLN is faster than *rebuild-from-LN* by 22-47%. This is because *rebuild-from-PLN* only scans the PLNs but *rebuild-from-LN* has to scan the entire LN list.

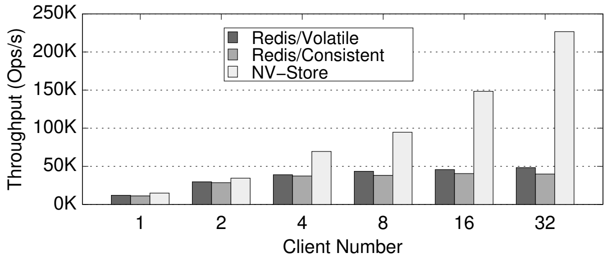
As the failure recovery of NV-Tree simply performs a *rebuild-from-LN*. The recovery time depends on the total number of LNs, but is bounded by the time of *rebuild-from-LN* as shown in Figure 10b.

To validate the consistency, we manually trigger the failure recovery by (1) killing NV-Tree process and (2) cutting the power supply when running both 100M insertion workload and YCSB workloads. Then we check whether NV-Tree has any data inconsistency or memory leak. We repeat these tests a few thousand times for NV-Tree and find it pass the check in all cases.

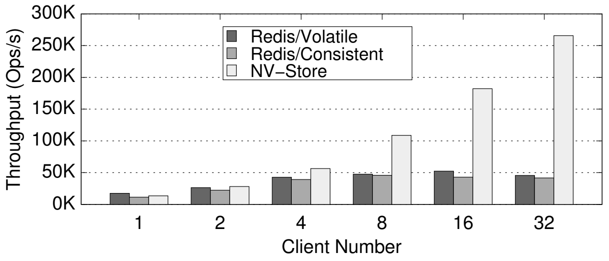
4.8 End-to-End Performance

In this subsection, we present the performance of our NV-Store under two YCSB workloads, StatusUpdate (read-latest) and SessionStore (update-heavy), compared to Redis. NV-Store is practically durable and consistent because it stores data in the NVM space directly allocated from NVDIMM using our modified system call. Redis can provide persistency by using `fsync` to write logs to an append-only file (AOF mode). With different `fsync` strategy, Redis can be either volatile if `fsync` is performed in a time interval, or consistent if `fsync` is performed right after each log write. We use the NVM space to allocate a RAMDisk for holding the log file so that Redis can be in-memory persistent. Note that it still goes through the POSIX interface (fsync).

Figure 11a shows the throughput of NV-Store and Redis under StatusUpdate workload which has 95%/5% search/insert ratio on keys chosen from a temporally weighted distribution to represent applications in which people update the online status while others view the latest status, which means newly inserted keys are preferentially chosen for retrieval. The result shows that NV-Store improve the throughput by up to 3.2X over both volatile and consistent Redis. This indicates the optimization of reducing cacheline flush for insertion can significantly improve the performance even with as low as 5% insertion percentage. Moreover, both volatile and



(a) Throughput of YCSB Workload: StatusUpdate



(b) Throughput of YCSB Workload: SessionStore

Figure 11: Throughput Comparison of NV-Store and Redis

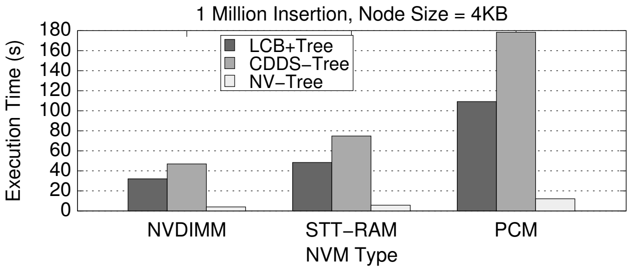


Figure 12: Execution Time on Different Types of NVM

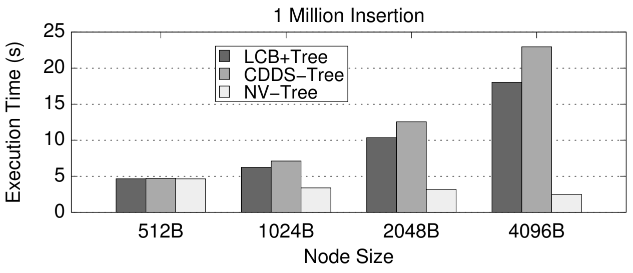


Figure 13: Estimated Execution Time with Epoch

consistent Redis are bottlenecked with about 16 clients while NV-Store can still scale up with 32 clients. The high scalability of NV-Store is achieved by (1) allowing concurrent search in LN while it is being updated, (2) searching in IN/PLNs without locks. Figure 11b shows the throughput under SessionStore workload which has 50%/50% search/update ratio on keys chosen from a Zipf distribution to represent applications in which people record recent actions. NV-Store can improve the throughput by up to 4.8X over Redis because the workload is more write-intensive.

4.9 Discussion

4.9.1 NV-Tree on Different Types of NVM

Given the write latency difference of NVDIMM (same as DRAM), PCM (180ns), STT-RAM (50ns) in Table 1, we explicitly add some delay before every memory write in our NV-Tree to investigate its performance on different types of NVM. Figure 12 shows the execution time of one million insertion in NV-Tree with 4KB nodes. Compared to the performance on NVDIMM, NV-Tree is only 5%/206% slower on STT-RAM/PCM, but LCB$^+$Tree is 51%/241% and CDDS-Tree is 87%/281% slower. NV-Tree suffers from less performance drop than LCB$^+$Tree and CDDS-Tree on slower NVM because of the reduction of CPU cacheline flush.

4.9.2 NV-Tree on Future Hardware: *Epoch* and *CLWB/CLFLUSHOPT/PCOMMIT*

Comparing to `MFENCE` and `CLFLUSH`, *epoch* and a couple of new instructions for non-volatile storage (`CLWB/CLFLUSHOPT/PCOMMIT`) added by Intel recently [26] are able to flush CPU cachelines without explicit invalidations which means it does not trigger any additional cache misses. As these approaches are still unavailable in existing hardware, we estimate LCB$^+$Tree, CDDS-Tree and our NV-Tree performance by removing the cost of L3 cache misses due to cacheline flushes the execution time (Figure 5a). For B$^+$Tree and volatile CDDS-Tree, such cost can be derived by deducting the number of L3 cache misses without cacheline flushes (Figure 1b) from that with cacheline flushes (Figure 9b). As shown in Figure 13, with the cache miss penalty removed, the performance improvement of NV-Tree over LCB$^+$Tree/CDDS-Tree is 7X/9X with 4KB nodes. This indicates our optimization of reducing cacheline flush is still valuable when flushing a cacheline without the invalidation becomes possible.

5 Conclusion and Future Work

In this paper, we quantify the consistency cost of applying existing approaches such as logging and versioning on B$^+$Tree. Based on our observations, we propose our NV-Tree which require the data consistency in NVM connected through a memory bus, e.g., NVM-based single level systems. By selectively enforcing consistency, adopting unsorted LN and organizing IN cache-optimized, NV-Tree can reduce the number of cacheline flushes under write-intensive workloads by more than 90% compared to CDDS-Tree. Using NV-Tree as the core data structure, we build a key-value store named NV-Store. Both NV-Tree and NV-Store are implemented and evaluated on a real NVDIMM platform instead of simulation. The experimental results show that NV-Tree outperforms LCB$^+$Tree and CDDS-Tree by up to 8X and 12X under write-intensive workloads, respectively. Our NV-Store increases the throughput by up to 4.8X under YCSB workloads compared to Redis. In our future work, we will continue to reduce the overhead of the rebuilding in larger datasets, validate and improve the performance of NV-Tree under skewed and TPC-C workloads, and explore NV-Tree in the distributed environment.

Acknowledgment

We would like to thank our teammates, Mingdi Xue and Renuga Kanagavelu, the anonymous reviewers and our shepherd, Nisha Talagala, for their helpful comments. This work was supported by Agency for Science, Technology and Research (A*STAR), Singapore under Grant No. 112-172-0010. Bingsheng's work was partly supported by a MoE AcRF Tier 2 grant (MOE2012-T2-1-126) in Singapore.

References

[1] AGIGATECH. Arxcis-nv (tm) non-volatile dimm. *http://www.vikingtechnology.com/arxcis-nv* (2014).

[2] BENDER, M. A., FARACH-COLTON, M., JOHNSON, R., , KRANER, R., KUSZMAUL, B. C., MEDJEDOVIC, D., MONTES, P., SHETTY, P., SPILLANE, R. P., AND ZADOK, E. Don't thrash: How to cache your hash on flash. In *Proceedings of the 38th International Conference on Very Large Data Bases (VLDB '12)* (Istanbul, Turkey, August 2012), Morgan Kaufmann.

[3] BERENSON, H., BERNSTEIN, P., GRAY, J., MELTON, J., O'NEIL, E., AND O'NEIL, P. A critique of ansi sql isolation levels. In *ACM SIGMOD Record* (1995), vol. 24, ACM, pp. 1–10.

[4] BHASKARAN, M. S., XU, J., AND SWANSON, S. Bankshot: caching slow storage in fast non-volatile memory. *Operating Systems Review 48*, 1 (2014), 73–81.

[5] BINGMANN, T. Stx b+ tree c++ template classes, 2008.

[6] CAULFIELD, A. M., DE, A., COBURN, J., MOLLOW, T. I., GUPTA, R. K., AND SWANSON, S. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *43rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2010, 4-8 December 2010, Atlanta, Georgia, USA* (2010), pp. 385–395.

[7] CAULFIELD, A. M., MOLLOV, T. I., EISNER, L. A., DE, A., COBURN, J., AND SWANSON, S. Providing safe, user space access to fast, solid state disks. *ACM SIGPLAN Notices 47*, 4 (2012), 387–400.

[8] CAULFIELD, A. M., AND SWANSON, S. Quicksan: a storage area network for fast, distributed, solid state disks. In *The 40th Annual International Symposium on Computer Architecture, ISCA'13, Tel-Aviv, Israel, June 23-27, 2013* (2013), pp. 464–474.

[9] CHEN, S., GIBBONS, P. B., AND NATH, S. Rethinking database algorithms for phase change memory. In *CIDR* (2011), pp. 21–31.

[10] CHIDAMBARAM, V., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Optimistic Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)* (Farmington, PA, November 2013).

[11] CHO, S., PARK, C., OH, H., KIM, S., YI, Y., AND GANGER, G. R. Active disk meets flash: A case for intelligent ssds. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing* (New York, NY, USA, 2013), ICS '13, ACM, pp. 91–102.

[12] COBURN, J., CAULFIELD, A. M., AKEL, A., GRUPP, L. M., GUPTA, R. K., JHALA, R., AND SWANSON, S. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2011), ASPLOS XVI, ACM, pp. 105–118.

[13] COMER, D. Ubiquitous b-tree. *ACM Computing Surveys (CSUR) 11*, 2 (1979), 121–137.

[14] CONDIT, J., NIGHTINGALE, E. B., FROST, C., IPEK, E., LEE, B., BURGER, D., AND COETZEE, D. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 133–146.

[15] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing* (2010), ACM, pp. 143–154.

[16] CULLY, B., WIRES, J., MEYER, D., JAMIESON, K., FRASER, K., DEEGAN, T., STODDEN, D., LEFEBVRE, G., FERSTAY, D., AND WARFIELD, A. Strata: High-performance scalable storage on virtualized non-volatile memory. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)* (Santa Clara, CA, 2014), USENIX, pp. 17–31.

[17] DEBRABANT, J., ARULRAJ, J., PAVLO, A., STONEBRAKER, M., ZDONIK, S., AND DULLOOR, S. R. A prolegomenon on oltp database systems for non-volatile memory. *Proceedings of the VLDB Endowment 7*, 14 (2014).

[18] DHIMAN, G., AYOUB, R., AND ROSING, T. Pdram: a hybrid pram and dram main memory system. In *Design Automation Conference, 2009. DAC'09. 46th ACM/IEEE* (2009), IEEE, pp. 664–669.

[19] DULLOOR, S. R., KUMAR, S., KESHAVAMURTHY, A., LANTZ, P., REDDY, D., SANKARAN, R., AND JACKSON, J. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), ACM, p. 15.

[20] EVERSPIN. Second generation mram: Spin torque technology. *http://www.everspin.com/products/second-generation-st-mram.html* (2004).

[21] FREITAS, R. F., AND WILCKE, W. W. Storage-class memory: The next storage system technology. *IBM Journal of Research and Development 52*, 4.5 (2008), 439–447.

[22] FRYER, D., SUN, K., MAHMOOD, R., CHENG, T., BENJAMIN, S., GOEL, A., AND BROWN, A. D. Recon: Verifying file system consistency at runtime. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)* (San Jose, CA, February 2012), pp. 73–86.

[23] GAO, S., XU, J., HE, B., CHOI, B., AND HU, H. Pcmlogging: Reducing transaction logging overhead with pcm. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management* (New York, NY, USA, 2011), CIKM '11, ACM, pp. 2401–2404.

[24] GRUPP, L. M., DAVIS, J. D., AND SWANSON, S. The bleak future of NAND flash memory. In *Proceedings of the 10th USENIX conference on File and Storage Technologies, FAST 2012, San Jose, CA, USA, February 14-17, 2012* (2012), p. 2.

[25] INTEL. Intel 64 and ia-32 architectures software developer's manual. *Volume 3A: System Programming Guide, Part 1* (2014).

[26] INTEL. Intel architecture instruction set extensions programming reference. *https://software.intel.com* (2014).

[27] JUNG, M., CHOI, W., SHALF, J., AND KANDEMIR, M. T. Triple-a: a non-ssd based autonomic all-flash array for high performance storage systems. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems* (2014), ACM, pp. 441–454.

[28] JUNG, M., WILSON III, E. H., CHOI, W., SHALF, J., AKTULGA, H. M., YANG, C., SAULE, E., CATALYUREK, U. V., AND KANDEMIR, M. Exploring the future of out-of-core computing with compute-local non-volatile memory. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis* (2013), ACM, p. 75.

[29] KAWAHARA, T. Scalable spin-transfer torque ram technology for normally-off computing. *IEEE Design & Test of Computers 28*, 1 (2011), 0052–63.

[30] KIM, H., SESHADRI, S., DICKEY, C. L., AND CHIU, L. Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)* (Santa Clara, CA, 2014), USENIX, pp. 33–45.

[31] Kim, M., Shin, J., and Won, Y. Selective segment initialization: Exploiting nvram to reduce device startup latency. In *Embedded Systems Letters* (2014), pp. 33–36.

[32] Kim, W.-H., Nam, B., Park, D., and Won, Y. Resolving journaling of journal anomaly in android i/o: Multi-version b-tree with lazy split. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)* (Santa Clara, CA, 2014), USENIX, pp. 273–285.

[33] Lee, D., and Won, Y. Bootless boot: Reducing device boot latency with byte addressable NVRAM. In *10th IEEE International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing, HPCC/EUC 2013, Zhangjiajie, China, November 13-15, 2013* (2013), pp. 2014–2021.

[34] Lee, E., Bahn, H., and Noh, S. H. Unioning of the buffer cache and journaling layers with non-volatile memory. In *FAST* (2013), pp. 73–80.

[35] Li, C., Shilane, P., Douglis, F., Shim, H., Smaldone, S., and Wallace, G. Nitro: a capacity-optimized ssd cache for primary storage. In *Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference* (2014), USENIX Association, pp. 501–512.

[36] Li, Y., He, B., Yang, R. J., Luo, Q., and Yi, K. Tree indexing on solid state drives. *Proceedings of the VLDB Endowment 3*, 1-2 (2010), 1195–1206.

[37] Liu, R.-S., Shen, D.-Y., Yang, C.-L., Yu, S.-C., and Wang, C.-Y. M. Nvm duet: unified working memory and persistent store architecture. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems* (2014), ACM, pp. 455–470.

[38] Lomet, D., and Salzberg, B. *Access methods for multiversion data*, vol. 18. ACM, 1989.

[39] Lv, Y., Cui, B., He, B., and Chen, X. Operation-aware buffer management in flash-based systems. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data* (2011), ACM, pp. 13–24.

[40] Mandelman, J. A., Dennard, R. H., Bronner, G. B., DeBrosse, J. K., Divakaruni, R., Li, Y., and Radens, C. J. Challenges and future directions for the scaling of dynamic random-access memory (dram). *IBM Journal of Research and Development 46*, 2.3 (2002), 187–212.

[41] Mármol, L., Sundararaman, S., Talagala, N., Rangaswami, R., Devendrappa, S., Ramsundar, B., and Ganesan, S. Nvmkv: A scalable and lightweight flash aware key-value store. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)* (2014), USENIX Association.

[42] Moraru, I., Andersen, D. G., Kaminsky, M., Tolia, N., Ranganathan, P., and Binkert, N. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *Proceedings of the 2013 Conference on Timely Results in Operating Systems* (2013).

[43] Mueller, W., Aichmayr, G., Bergner, W., Erben, E., Hecht, T., Kapteyn, C., Kersch, A., Kudelka, S., Lau, F., Luetzen, J., et al. Challenges for the dram cell scaling to 40nm. In *Electron Devices Meeting, 2005. IEDM Technical Digest. IEEE International* (2005), IEEE, pp. 4–pp.

[44] Narayanan, D., and Hodson, O. Whole-system persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2012), ASPLOS XVII, ACM, pp. 401–410.

[45] Pelley, S., Chen, P. M., and Wenisch, T. F. Memory persistency. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on* (2014), IEEE, pp. 265–276.

[46] Pillai, T. S., Chidambaram, V., Alagappan, R., Al-Kiswany, S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. 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 '14)* (Broomfield, CO, October 2014).

[47] Qin, D., Brown, A. D., and Goel, A. Reliable writeback for client-side flash caches. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (Philadelphia, PA, June 2014), USENIX Association, pp. 451–462.

[48] Qureshi, M. K., Srinivasan, V., and Rivers, J. A. Scalable high performance main memory system using phase-change memory technology. *ACM SIGARCH Computer Architecture News 37*, 3 (2009), 24–33.

[49] 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., et al. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development 52*, 4.5 (2008), 465–479.

[50] Sanfilippo, S., and Noordhuis, P. Redis. *http://redis.io* (2009).

[51] Seshadri, S., Gahagan, M., Bhaskaran, S., Bunker, T., De, A., Jin, Y., Liu, Y., and Swanson, S. Willow: A user-programmable ssd. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), USENIX Association, pp. 67–80.

[52] Sriram Subramanian, Swami Sundararaman, Nisha Talagala, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. Snapshots in a Flash with ioSnap. In *EuroSys '14* (Amsterdam, Netherlands, April 2014).

[53] Venkataraman, S., Tolia, N., Ranganathan, P., Campbell, R. H., et al. Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST* (2011), pp. 61–75.

[54] Volos, H., Nalli, S., Panneerselvam, S., Varadarajan, V., Saxena, P., and Swift, M. M. Aerie: flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), ACM, p. 14.

[55] Volos, H., Tack, A. J., and Swift, M. M. Mnemosyne: Lightweight persistent memory. In *ACM SIGARCH Computer Architecture News* (2011), vol. 39, ACM, pp. 91–104.

[56] Vučinić, D., Wang, Q., Guyot, C., Mateescu, R., Blagojević, F., Franca-Neto, L., Moal, D. L., Bunker, T., Xu, J., Swanson, S., and Bandić, Z. Dc express: Shortest latency protocol for reading phase change memory over pci express. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)* (Santa Clara, CA, 2014), USENIX, pp. 309–315.

[57] Wang, C., Vazhkudai, S. S., Ma, X., Meng, F., Kim, Y., and Engelmann, C. Nvmalloc: Exposing an aggregate ssd store as a memory partition in extreme-scale machines. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International* (2012), IEEE, pp. 957–968.

[58] Wu, X., and Reddy, A. Scmfs: a file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (2011), ACM, p. 39.

[59] YIYING ZHANG, ANDREA C. ARPACI-DUSSEAU, REMZI H. ARPACI-DUSSEAU. Warped Mirrors for Flash. In *Proceedings of the 29th IEEE Conference on Massive Data Storage (MSST '13)* (Long Beach, California, May 2013).

[60] YIYING ZHANG, LEO ARULRAJ, ANDREA C. ARPACI-DUSSEAU, REMZI H. ARPACI-DUSSEAU. De-indirection for Flash-based SSDs with Nameless Writes. In *Proceedings of the 10th Conference on File and Storage Technologies (FAST '12)* (San Jose, California, February 2012).

[61] ZHOU, P., ZHAO, B., YANG, J., AND ZHANG, Y. A durable and energy efficient main memory using phase change memory technology. In *ACM SIGARCH Computer Architecture News* (2009), vol. 37, ACM, pp. 14–23.

Towards SLO Complying SSDs Through OPS Isolation

Jaeho Kim[1], Donghee Lee[1], and Sam H. Noh[2]

[1]University of Seoul, {kjhnet10, dhl_express}@uos.ac.kr
[2]Hongik University, http://next.hongik.ac.kr

Abstract

Virtualization systems should be responsible for satisfying the service level objectives (SLOs) for each VM. Performance SLOs, in particular, are generally achieved by isolating the underlying hardware resources among the VMs. In this paper, we show through empirical evaluation that performance SLOs cannot be satisfied with current commercial SSDs. We show that garbage collection is the source of this problem and that this cannot be easily controlled because of the interaction between VMs. To control the effect of garbage collection on VMs, we propose a scheme called OPS isolation. OPS isolation allocates flash memory blocks so that blocks of one VM do not interfere with blocks of other VMs during garbage collection. Experimental results show that performance SLO can be achieved through OPS isolation.

1 Introduction

The use of flash memory based Solid State Drives (SSDs) is now commonplace and is being extended to server virtualization [1, 2]. Virtualization systems should be responsible for satisfying the service level objective (SLO) for each VM. Performance service level objectives (SLOs), in particular, are generally achieved by isolating the underlying hardware resources among the VMs. Consequently, many studies for allocating the resources for each VM have been conducted and existing products such as VMware ESX server hypervisor that provide isolated CPU and memory are available [3, 4].

Recent studies making use of SSDs as a shared cache resource among virtual machines (VM) in virtualization systems have been conducted [1, 2]. In this work, we revisit this issue, first, by quantitatively examining IO performance and interference among the VMs within the SSD. We show that depending on the status of the SSD, experimental results can vary significantly, and this difference comes from the interference among the VMs. We then propose *OPS (Over-Provisioning Space) Isolation* at the FTL (Flash Translation Layer) layer such that the OPS of each VM is isolated from being affected by other VMs. We show that performance SLOs of VMs can be satisfied through OPS isolation.

Table 1: Characteristics of IO workloads

Workload	Request Total	Write Ratio	Average Write Size
Financial	7.1GB	0.76	14KB
MSN	14.6GB	0.96	27KB
Exchange	9.8GB	0.67	17KB

The rest of the paper is organized as follows. In the next section, we present the motivation of this work and work related to this study. In Section 3, we look into the internals of SSDs to understand the effects of garbage collection on concurrently executing VMs. In Section 4, we present OPS isolation, the main contribution of this work along with performance evaluations. Finally, in Section 5, we give a summary and conclude.

2 Motivation and Related Work

As motivation, we conduct a set of experiments and observe the performance results that are returned. We show that the performance reported by SSDs vary widely even when executing the same workloads and that the performance of SSDs are strongly affected by their state, which is difficult to control. The results serve as motivation to develop SSDs that are performance predictable.

All experiments in this section are conducted using a commercial SSD that is purchased off-the-shelf. The product uses MLC-based flash memory with a capacity of 128GB. The experiments conducted start from either a clean state or an aged state. Aging is conducted by issuing random writes (including overwrites) of sizes ranging from 4KB through 32KB for a total write that exceeds the SSD capacity. As the SSD becomes full, the SSD becomes busy performing garbage collection. We consider an SSD at this state to be an aged SSD.

Table 2: KVM environment

Description	Host	VM-1∼4
CPU core	8	1
Memory size	32GB	1GB
OS	Ubuntu-14.x with KVM	Ubuntu-14.x
Storage	Dedicated storage	Each 30GB SSD

Three workloads, specifically, Financial, MSN, and Exchange, are used in the experiments. The details regarding the characteristics of the workloads are shown in Table 1. The original workloads used here are traces provided by UMass Trace Repository and MSR [5, 6]. As the experiments in this section use real SSDs, we require real IO requests. Hence, we make use of a replayer tool that takes requests from the trace and turns them into real requests to the device [7]. For each request, a single threaded replayer waits for it to complete, upon which various statistics are gathered.

2.1 Effect of SSD aging

We conduct a set of experiments to show how VMs are affected by various conditions of the storage system. First, we show how proportionality varies as the state of the SSD varies. Our goal is to proportionally distribute IO usage of a shared SSD among VMs. To do so, we first create kernel-based virtual machine (KVM) VMs with the same workloads. The VM settings and the rest of the environment for the experiments are summarized in Table 2. We use the Cgroup [8] Linux feature that limits, accounts, and isolates hardware resource usage of process groups to assign different weights to the VMs (allocating higher throughput for higher weights). We then measure IO performance of each VM.

The results in Figure 1 show that for all the workloads, on the HDD, proportionality is close to the IO weight except for VM-10. However, for the SSDs, proportionality deviates. Note that deviation is worse for the aged SSD than the clean SSD. One can conjecture that this is due to garbage collection (GC), which is largely considered to be the bandit of all wrong in SSDs. Indeed, we show later that during GC, VMs are actually moving other VM's data around, which is unnecessary data movement from the GC triggering VM's point of view, resulting in inaccurate performance control. Another observation from Figure 1 is that the effect of GC on the various VMs is not uniform. That is, we understand that GC is affecting performance in a negative way, but how each VM is affected is not clear.

2.2 Effect of concurrent execution

We perform another set of experiments, this time with a mix of workloads. Four sets of results are presented in Figure 2, and we discuss how they were obtained and

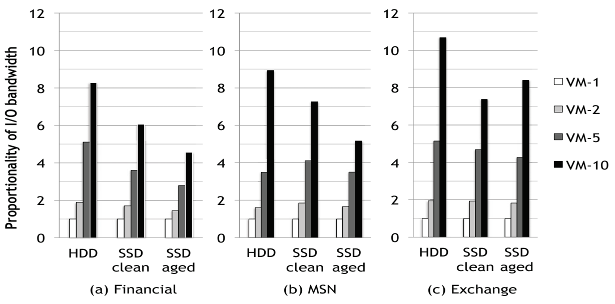


Figure 1: IO bandwidth with Cgroup relative to VM-1 for various workloads. (Notation: VM-x, where x is weight value.)

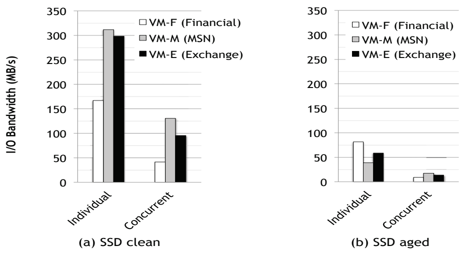


Figure 2: IO bandwidth of individual and concurrent execution of VMs.

what they imply. Figure 2(a) shows the observed bandwidth with a clean SSD. The individual results are obtained by executing each workload starting from a clean SSD for each workload. The concurrent results are obtained by executing the three workloads concurrently on a clean SSD. The three workloads are exactly the same for both individual and concurrent executions, so the total footprint is also the same.

We observe from the results, however, that concurrent execution performs markedly worse than executing each VM individually. With concurrent execution, each VM performance is roughly a third of each individual execution with some deviation among the individual VMs. We also observe that bandwidth is not being consumed in full with the total bandwidth consumed by the three concurrent workloads being roughly 270MB/s.

The results for Figure 2(b) were obtained in a manner similar to that of the clean SSD, only that the SSD goes through an aging process that was described earlier. Three points are noteworthy regarding these results. First, the overall performance drop is significant. Again, the culprit will be GC. Second, we also see that with concurrent execution the performance drop is significant compared to individual execution as was observed with the clean SSD, but the drop is more significant. Finally, and more importantly, the effect of aging on the individual VMs varies considerably depending on the VM. That is, the effect of aging is not uniform. For example, for the individual execution, while the bandwidth of VM-F

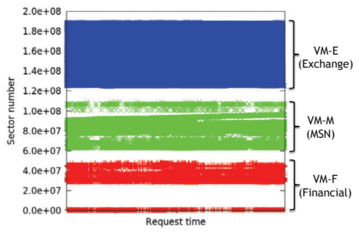


Figure 3: Sector access pattern of the Financial, MSN, and Fileserver workloads.

Table 3: Parameters of SSD simulator

Parameter	Description
Page size	4KB
Block size	512KB
Page read	60us
Page write	800us
Block erase	1.5ms
Page Xfer latency	102us

is reduced by only half, for VM-M, observed bandwidth is reduced to only 15% of the clean SSD case.

Again, we reach the same conclusion as in the previous subsection. That is, we point our fingers at GC for the reduction in performance. However, the effect of GC on individual VMs is not at all uniform. We know GC have negative effects, but how the VMs are being affected is not clear.

2.3 Related Work

In virtualization systems, service level objectives (SLOs) for VMs is achieved through transparent allocation of resources for each VM. Products such as VMware ESX server hypervisor provides isolated CPU and memory to satisfy SLOs [3, 4]. Numerous studies have been conducted to satisfy SLOs for VMs [1, 2, 9, 10, 11, 12, 13, 14]. In particular, DeepDive identifies and manages performance interference between VMs sharing hardware resources [11]. It is regarded as the first end-to-end system that handles interference of major resources such as CPU, memory, and IO.

Studies to provide IO SLOs among VMs have also been conducted [9, 10, 14]. mClock provides proportional-share fairness among the VMs through IO scheduling of the hypervisor [10]. Research on allocation of a shared SSD cache for VMs have also been conducted [1, 2]. S-CAVE effectively manages a shared SSD cache by using runtime information among VMs [1]. vCacheShare addresses the allocation decision for server flash cache (SFC) based on IO access characteristics of running VMs [2]. The goal of their work is in maximizing the utilization of the SSD cache and achieving performance isolation. Our work shows that controlling the SSD from outside the SSD is difficult as one cannot

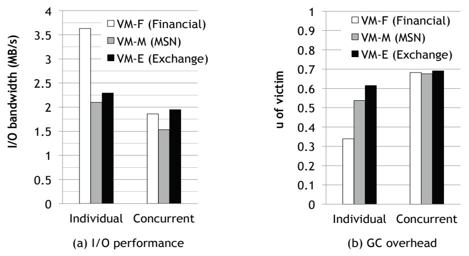


Figure 4: (a) IO performance and (b) GC overhead with SSD simulator

control the internal workings of GC.

A recent study called the Multi-streamed SSD proposes a technique similar to what we propose [15]. Here, Kang et al. propose to make changes to the block device interface to manage blocks based on what they call streams, that is, blocks with similar expected lifetime. This work is different from ours in that their focus is on maximizing the overall performance of SSDs through workload independent block characterization, while we concentrate on controlling each VM within an SSD for SLO compliance.

3 Understanding the Effect of GC

In this section, we first discuss the effect of GC on individual workloads when workloads are run concurrently. This is done using a simulation environment. Then, we present experimental results that imply that commercial SSDs have similar effects.

3.1 GC effect on concurrent workloads

To analyze GC overhead with concurrent workloads, we conduct experiments with SSD extension for DiskSim as the internal workings can be monitored. DiskSim employs a page-mapped FTL used in most SSD products. As for GC, it uses a greedy policy to select the victim block when the number of free blocks drops below a certain threshold. Other parameters of the simulator are presented in Table 3. Also, we use the same workloads as the previous section, and to take into account the VM nature of the previous experiments, the traces that we use for these experiments are those captured as the experiments are performed in Section 2. Fig. 3 shows the sector access patterns for the workloads. The figure shows distinct bands of space being accessed by each workload.

Figure 4(a) shows the IO performance for cases where the workloads are executed individually and concurrently, similarly to those described in Section 2. Here, we age the DiskSim simulator in similar fashion as the commercial SSD before the performance is measured. For the individually run case, the trend in performance is similar to those obtained for the real SSD. For the concurrent case, the trend is quite different, but this is

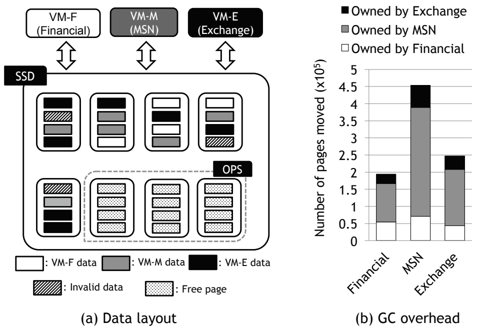


Figure 5: (a) Data layout of concurrent workloads in conventional SSD and (b) number of pages moved for each workload during GC.

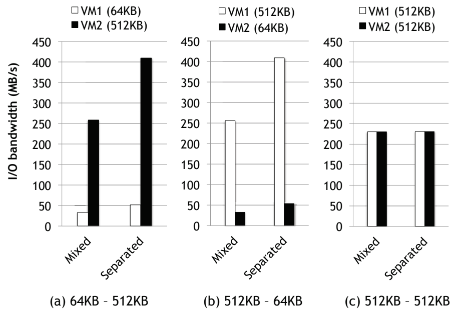


Figure 6: IO bandwidth of VMs generating synthetic requests on a commercial SSD after the 'Mixed' and 'Separated' initialization steps. With 'Mixed', initialization is done with VM1 and VM2 workloads executed concurrently, while for 'Separated', initialization is done by first executing VM1, followed by VM2 execution.

expected as the FTL employed will be different and the three workloads are simultaneously affecting the FTL in various ways. Note, however, that though the exact performance trend may be different, the performance drop of the individual workload varies as was observed for the commercial SSD.

Let us now turn to the reason behind this observation. For this, we observe the internal status and movements of the pages within the device. This is done by tagging each page (in the OOB (Out-Of-Band) area) with the ID of the particular VM that instigated the request and monitoring the tags as the experiments are conducted.

Figure 4(b) shows the average number of valid pages (denoted u, for utilization) of the victim blocks selected for GC. This value is lower when each workload is executed individually than when the workloads are executed concurrently. In particular, the difference in u is proportional to the difference in performance observed in Figure 4(b), that is, largest for Financial and smallest for Exchange. This is to say that while each workload is being negatively influenced by each other as they execute concurrently, Financial is being influenced the most.

The reason behind this negative influence can be explained through Figure 5. Figure 5(a) shows a data layout of a typical SSD when requests from multiple workloads arrive concurrently. The FTL takes each page and randomly places them among the available blocks. Consequently, blocks contain pages from various workloads. Hence, upon an erase while servicing a particular workload, live pages from other workloads in the victim block will be moved to a new block during the GC process.

Figure 5(b) shows the number of pages that are moved during GC for each workload. The pages are distinguished by the owner of the page when they are moved. For example, of the 190K number of pages moved while executing the Financial workload, only 30% of them are those of its own. This says that though GC is a necessity, much of the work involved in the GC process are actually unnecessary work induced by other workloads. Then the solution to this problem is to find a means to isolate the GC process so that GC for each workload does not interfere with other workloads.

3.2 Observation in commercial SSDs

In the previous subsection, we showed results that alluded to the interfering phenomenon using the DiskSim simulation environment. Though simulations are the basis of many important studies and innovations, one still has to wonder if what we observed in the previous subsection actually occurs in real SSDs. To verify this, we perform the following set of experiments.

Taking the commercial SSD that we used previously, we create two VMs, VM1 and VM2 that generates writes and over-writes of 64KB and 512KB sized random requests, which represent small and large requests, respectively. The choice of the two sizes is to vary the mix of data within blocks as will be described below. Hence, the results that we show do not vary for different size choices so long as the two sizes differ by some significant value.

With the two VMs, we perform two different experiments. In the first, starting from a clean SSD, the two VMs are run simultaneously as an initialization step for some amount of time. As a result, the SSD will be populated with data from VM1 and VM2 resulting in data from the two VMs being intermixed. Then, the VMs are run again, but this time the performance is measured and is reported as 'Mixed' in Figure 6(a). In the second set of experiments, we also go through an initialization process but this time the VMs are run one at a time filling in the same amount of data as before. This time, because of the request size and as the VMs are run in sequence, the FTL will (generally) not place data from the two VMs within the same block. Then, the VMs are run and the performance measured. The results for these are reported as 'Separated' in Figure 6(a). Note that the 'Separated' scenario performs substantially better than 'Mixed'.

The results shown in Figure 6(b) are results from the

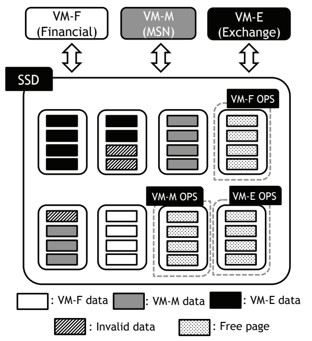


Figure 7: Sample data layout with OPS isolation

same experiments with only the order of initialization (512KB first, then 64KB random writes) for the 'Separate' result being different. Figure 6(c) shows the results for the same sequence of experiments, but with same sized (512KB) random writes. These results are shown to contrast them to those of Figures 6(a) and (b).

The reason the 'Separated' scenario performs substantially better than the 'Mixed' scenario is likely because in the 'Separated' scenario the pages from one VM do not negatively influence the other VM. Though we cannot be definite regarding the workings of the SSD due to their propriety nature, these results are in line with the findings of the DiskSim evaluation.

4 SLO Complying SSD and Its Performance Evaluation

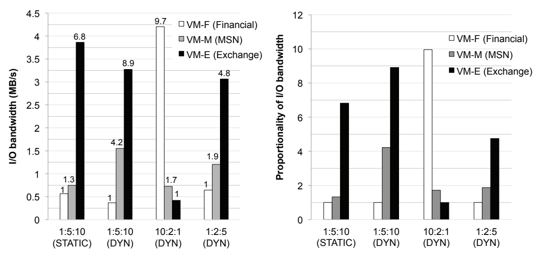
In this section, we discuss how the negative effect of garbage collection can be mitigated so that SLO requests may be satisfied. We start by reviewing previous work that formulates IO performance of flash memory based SSDs. We propose OPS isolation as a means to control the performance of individual VMs. Experimental results showing that performance proportionality of VMs can be obtained through OPS isolation are presented.

4.1 Calculating IOPS of SSD

To guarantee IO performance SLO among VMs sharing an SSD, we need to understand the relation between IO performance and the GC overhead. Performance characterizations of NAND flash memory SSDs have been studied extensively, and from these it is well understood that write IO performance can be represented as shown in Equation 1 [16, 17].

$$IOPS_{SSD_W} = \frac{1}{t_{GC} + t_{PRG} + t_{Xfer}} \tag{1}$$

where t_{PRG} and t_{Xfer} are constant values determined by the flash chip manufacturers representing the time to program a page and the time to transfer a page, respectively, and t_{GC}, which is the time to GC defined as $t_{GC} = WAF(u) \cdot t_{PRG}$. $WAF(u)$, which stands for Write



(a) I/O bandwidth (b) I/O proportionality

Figure 8: Evaluation results

Amplication Factor and which is a function of u, the utilization of the flash memory blocks, refers to the additional page writes caused by GC to service the write requests [16]. Studies have shown that $WAF(u)$ can be represented as shown in Equation 2 where N_p is the number of pages per block. Note that u can be measured from the SSD or can be estimated from the ratio of the user data and initial OPS size [18, 19]. Also note, however, u is a value that represents the entire SSD.

$$WAF(u) = \frac{u \cdot N_P}{(1-u) \cdot N_P} = \frac{u}{(1-u)} \tag{2}$$

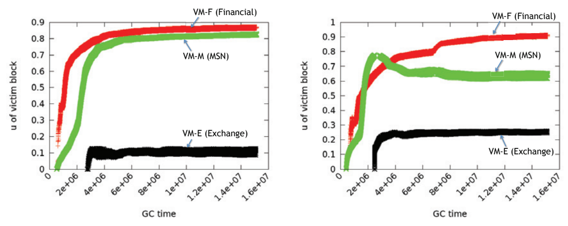
From Equations 1 and 2, we know that write performance of SSDs is determined by the GC overhead, which is determined by u, which in turn, is determined by the OPS [20]. Therefore, to control IO performance, managing OPS properly is imperative.

Typically, OPS is globally managed in SSDs. Hence, VM based IO performance guarantees are difficult, if not impossible, to handle. We propose to isolate OPS handling so that OPS is managed per VM. This allows more manageable control over IO performance for each VM.

4.2 OPS isolation

To satisfy SLO requests from VMs, we propose to dedicate flash memory blocks, including OPS, to each VM separately when allocating pages to VMs so that interference can be prevented during GC. Figure 7 shows an example of how blocks would be allocated among the three VMs concurrently requesting flash space. Contrasting this figure with that of Figure 5(a) shows how the two differ. Observe in Figure 7 all blocks consists of free pages or pages from only one VM. As OPS is also dedicated to a single VM, write requests from each VM will be placed only within the same block preventing pages from different VMs from being mixed.

To satisfy SLO requests, IO performance must also be guaranteed. As discussed in Section 4.1, performance is eventually influenced by the OPS allocated to each VM. Algorithm 1 presents the algorithm that we use to partition the OPS among the competing concurrent VMs. For this study, we simply take the proportional division of the total possible IOPS as satifying the SLO request.



(a) u with Static (b) u with Dynamic

Figure 9: Average u of victim block along GC time

Initially, the flash memory blocks, including the OPS, are partitioned among the competing VMs based on the weight requested. Using the IOPS specified for the SSD, we calculate the estimated IOPS of each VM based on the requested weight. Then, a separate u for each VM can be calculated using Equation 1. After this initialization phase (lines 3 through 11), the OPS size is dynamically and periodically adjusted to maintain the IOPS that was initially designated (lines 12 through 20). For example, take 3 VMs, A, B, and C that are given target proportional IO performance weights of 1, 3, and 6 with higher weights being allotted higher bandwidth. The initial OPS size for each VM is set by the ratio of the weights (lines 3 and 4), that is, 10%, 30%, and 60% of the total OPS is allotted to VMs A, B, C, respectively. If we assume that the specified IOPS for the SSD is 1000, the target IOPS is also set by the weight designated for each VM (line 9). Finally, the target utilization is set for each VM using Equation 1 (line 11). Then, utilization is monitored so that the OPS size can be adjusted if the utilization drifts from the target utilization. This adjusting is done before every GC with the OPS size increased or decreased by one block when necessary.

To implement features such as this in an SSD, the storage interface must change. Recent studies such as the Multi-streamed SSD [15] have proposed changes to the interface for enhanced performance benefits. Similarly, our method requires minimal information, such as a tag identifying the workload, which is already provided with eMMC flash [21], and the SLO requirement such as weight, to be transferred to the SSD.

4.3 Performance Evaluation

To evaluate the SLO complying SSD that we propose, we implement Algorithm 1 in the DiskSim SSD extension [22] that we used in Section 3. For the workloads, we again use the same traces that were previously used.

Figure 8(a) shows the results for the various workloads as the weight of each VM is given differently. In the figure, the x-axis shows groups of VMs that are executed concurrently with the weights allotted to the VMs. For the static case, only lines 3 and 4 of Algorithm 1 are executed and the OPS size does not change throughout the execution. For the rest of the results, the OPS size is

Algorithm 1 OPS Allocation

```
 1: //N: Number of VMs running concurrently
 2: //W($VM_i$) refers to weight given to $VM_i$
 3: **for** each $VM_i$ **do** //Initialize OPS size for $VM_i$
 4:     OPS($VM_i$) $\leftarrow$ $OPS_{total}$ $\times$ Ratio of W($VM_i$)
 5: //Use SSD IOPS value
 6: $IOPS_{total}$ $\leftarrow$ SSD IOPS specification
 7: **for** each $VM_i$ **do**
 8:     //Divide total IOPS according to $VM_i$ weight
 9:     IOPS($VM_i$) $\leftarrow$ $IOPS_{total}$ $\times$ Ratio of W($VM_i$)
10:     //Find $u$ for each $VM_i$ using Equation 1
11:     $u$(IOPS($VM_i$)) $\leftarrow$ Equation 1
12: **Begin Do** periodically adjust OPS:
13: **for** each $VM_i$ **do**
14:     //Otherwise if current utilization is higher
15:     **if** $u$(Cur($VM_i$)) > $u$(IOPS($VM_i$)) **then**
16:         Increase OPS($VM_i$)
17:         //Find $VM_i$ with max current utilization
18:         $VM_i$ $\leftarrow$ Max(VM($u$(Cur($VM_i$))))
19:         Decrease OPS($VM_i$)
20: **End Do**
```

dynamically adjusted according to Algorithm 1. The y-axis represents the absolute bandwidth achieved and the numbers on top of each bar represents the performance ratio relative to the bar with the smallest weight. For easy comparison, Figure 8(b) shows the same results in Figure 1 format.

The results show that using OPS isolation and dynamically adjusting the OPS size based on u results in quite accurate proportionality of IO bandwidth. However, static OPS isolation is not effective as there is no leeway to adjust the OPS size according to the workload characteristics.

Figure 9(a) shows how u changes when OPS is set to a static value determined by the proportional weight of the VMs. In contrast, Figure 9(b) shows u changing when the whole of Algorithm 1 is employed, dynamically adjusting the OPS size as need be.

5 Conclusion

In this paper, we showed that performance SLOs cannot be satisfied with current commercial SSDs because of garbage collection interference among competing virtual machines (VM). To resolve this problem, we proposed OPS isolation, a scheme that allocates flash memory blocks in such a way that blocks are not shared among VMs, but are wholly dedicated to each individual VM. Our experimental results showed that OPS isolation is an effective way for SSDs to provide performance SLOs to competing VMs.

Acknowledgement

We would like to thank our shepherd John Strunk and the anonymous referees for their constructive comments that helped to improve the presentation of the paper. We also thank Yongseok Oh for providing us with the trace replayer and Sooyeong Bae for helping with some of the experiments. This research was supported in part by Seoul Creative Human Development Program funded by Seoul Metropolitan Government(No. HM120006), by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MEST) (No. 2012R1A2A2A01045733), and by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education, Science and Technology(2010-0025282).

References

[1] Tian Luo, Siyuan Ma, Rubao Lee, Xiaodong Zhang, Deng Liu, and Li Zhou. S-CAVE: Effective SSD Caching to Improve Virtual Machine Storage Performance. In *Proc. of International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 103–112, 2013.

[2] Fei Meng, Li Zhou, Xiaosong Ma, Sandeep Uttamchandani, and Deng Liu. vCacheShare: Automated Server Flash Cache Space Management in a Virtualization Environment. In *Proc. of USENIX Conference on USENIX Annual Technical Conference (ATC)*, pages 133–144, 2014.

[3] VMware Inc. Distributed Resource Scheduler. `http://www.vmware.com/files/pdf/VMware-Distributed-Resource-Scheduler-DRS-DS-EN.pdf`.

[4] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, 2002.

[5] UMASS TRACE REPOSITORY. OLTP Application I/O. http://traces.cs.umass.edu, 2002.

[6] Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, and Antony Rowstron. Migrating Server Storage to SSDs: Analysis of Tradeoffs. In *Proc. of ACM European Conference on Computer Systems (EuroSys)*, pages 145–158, 2009.

[7] Yongseok Oh. Trace-replay. `https://bitbucket.org/yongseokoh/trace-replay`.

[8] Paul Menage. CGROUPS. `https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt`.

[9] Ajay Gulati, Irfan Ahmad, and Carl A. Waldspurger. PARDA: Proportional Allocation of Resources for Distributed Storage Access. In *Proc. of USENIX Conference on File and Storage Technologies (FAST)*, pages 85–98, 2009.

[10] Ajay Gulati, Arif Merchant, and Peter J. Varman. mClock: Handling Throughput Variability for Hypervisor IO Scheduling. In *Proc. of USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 1–7, 2010.

[11] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In *Proc. of USENIX Conference on Annual Technical Conference (ATC)*, pages 219–230, 2013.

[12] R. Prabhakar, S. Srikantaiah, C. Patrick, and M. Kandemir. Dynamic Storage Cache Allocation in Multi-Server Architectures. In *Proc. of Conference on High Performance Computing Networking, Storage and Analysis (SC)*, pages 8:1–8:12, 2009.

[13] David Shue, Michael J. Freedman, and Anees Shaikh. Performance Isolation and Fairness for Multi-tenant Cloud Storage. In *Proc. of USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 349–362, 2012.

[14] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. IOFlow: A Software-defined Storage Architecture. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)*, pages 182–196, 2013.

[15] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. The Multi-streamed Solid-State Drive. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2014.

[16] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. Write Amplification Analysis in Flash-based Solid State Drives. In *Proc. ACM International Ststems and Storage Conference (SYSTOR)*, pages 10:1–10:9, 2009.

[17] Yongseok Oh, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Caching Less for Better Performance: Balancing Cache Size and Update Cost of Flash Memory Cache in Hybrid Storage Systems. In *Proc. of USENIX Conference on File and Storage Technologies (FAST)*, pages 313–326, 2012.

[18] Hunki Kwon, Eunsam Kim, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Janus-FTL: Finding the Optimal Point on the Spectrum between Page and Block Mapping Schemes. In *Proc. International Conference on Embedded Software (EMSOFT)*, pages 169–178, 2010.

[19] Wenguang Wang, Yanping Zhao, and Rick Bunt. HyLog: A High Performance Approach to Managing Disk Layout. In *Proc. of USENIX Conference on File and Storage Technologies (FAST)*, pages 145–158, 2004.

[20] Radu Stoica and Anastasia Ailamaki. Improving Flash Write Performance by Using Update Frequency. *Proc. VLDB Endowment*, 6(9):733–744, 2013.

[21] JEDEC. Data Tag Mechanism of eMMC, JEDEC Standard Specification No. JESD84-B45. `http://www.jedec.org/sites/default/files/docs/jesd84-B45.pdf`.

[22] Vijayan Prabhakaran and Ted Wobber. SSD Extension for DiskSim Simulation Environment. `http://research.microsoft.com/en-us/downloads/b41019e2-1d2b-44d8-b512-ba35ab814cd4`, 2009.

Boosting Quasi-Asynchronous I/O for Better Responsiveness in Mobile Devices

Daeho Jeong[*+], Youngjae Lee[+], and Jin-Soo Kim[+]

[*]*Samsung Electronics Co., Suwon, Korea* [+]*Sungkyunkwan University, Suwon, Korea*

Abstract

Providing quick system response for mobile devices is of great importance due to their interactive nature. However, we observe that the latency of file system operations increases dramatically under heavy asynchronous I/Os in the background. A careful analysis reveals that most of the delay arises from an unexpected situation where the file system operations are blocked until one or more asynchronous I/O operations are completed. We call such an I/O – which is issued as an asynchronous I/O but has the property of a synchronous I/O as some tasks are blocked on it – *Quasi-Asynchronous I/O (QASIO)*.

We classify the types of dependencies between tasks and QASIOs and then show when such dependencies occur in the Linux kernel. Also, we propose a novel scheme to detect QASIOs at run time and boost them over the other asynchronous I/Os in the I/O scheduler. Our measurement results on the latest smartphone demonstrate that the proposed scheme effectively improves the responsiveness of file system operations.

1 Introduction

Mobile devices such as smartphones and tablet PCs have become one of the most popular consumer electronics devices. According to the Gartner's survey, the worldwide shipments of mobile devices are estimated at 2 billion units in 2013 and are expected to be nine times higher than those of traditional PCs by 2015 [7].

A key feature in providing satisfactory user experiences on mobile devices is fast responsiveness. Since most applications running on mobile devices interact with users all the time, quick system response without any noticeable delay is of great importance. In spite of the increase in storage capacity and significant improvement in its performance, storage is still often blamed for impairing end-user experience in mobile devices [9].

Several efforts have been made to understand the I/O characteristics of popular mobile applications and their implications on the underlying storage media, NAND flash memory [9, 12]. Based on such investigations, various optimizations have been proposed for the I/O stack of the mobile platform including file systems, I/O schedulers, and block layers [10, 8, 11, 18]. However, previous approaches mostly view the problem from the perspective of increasing throughput or enhancing the lifetime of NAND flash memory.

In this paper, we focus on the latency of file system operations such as `creat()`, `write()`, `truncate()`, `fsync()`, etc. under heavy I/O load. We observe that when the system has lots of I/O requests issued asynchronously, the latency of these file system operations increases dramatically so that the responsiveness of applications is severely degraded. In our evaluations with one of the latest Android-based smartphones, the time to launch an application has been slowed down by a factor of 2.4 in the worst case when a large amount of file writes is in progress in the background.

This problem is projected to become worse in the future as the peripherals of mobile devices continue to adopt newer and more advanced technology. For example, the latest smartphones are equipped with Wi-Fi 802.11ac (1Gbps) and USB v2.0 (480Mbps) modules which can generate the I/O traffic of several tens of megabytes per second. It is very likely for a user to run an application while downloading some large files in the background through Wi-Fi or USB connections. In this case, the responsiveness of the foreground task will be affected significantly by massive asynchronous I/O operations.

Some degree of delay is inevitable when the foreground task accesses files under heavy asynchronous I/Os as long as they share the same storage device. Surprisingly, however, we find out that most of the delay arises from an unexpected circumstance where the file system operations are *unintentionally* blocked until one or more *asynchronous* I/O operations are finished. This phenomenon contradicts the conventional wisdom that an asynchronous I/O operation can be performed at any time as no one waits for it. Since asynchronous I/Os have lower priority than synchronous I/Os and handling of asynchronous I/Os is optimized not for latency but for throughput, the responsiveness of the foreground task can be highly affected when it has to wait for the completion of asynchronous I/Os. Our measurement with a high-end smartphone shows that a single invocation of a seemingly benign `creat()` or *buffered* `write()` system call can take more than 1 second, when its execution is blocked due to pending asynchronous I/Os (cf. Table 2). It is common that the worst case delay of these system calls increases to several seconds for low-end smartphones with worse storage performance.

To address this problem, we introduce a new type of I/O operations called *Quasi-Asynchronous I/O (QASIO)*. QASIOs are defined as the I/O operations which are issued asynchronously, but should be treated as synchronous I/Os since other tasks are blocked on them. Note that some of asynchronous I/Os are promoted to QASIOs *at run time* when a task gets blocked on them. Since the execution of the blocked task depends on QASIOs, QASIOs should be prioritized over (true) asynchronous I/Os for better responsiveness. To the best of our knowledge, this work is the first study to discuss the dependency between file system operations and asynchronous I/O operations.

We propose a novel scheme to detect QASIOs and boost them in the Linux kernel. First, we analyze three problematic scenarios where the responsiveness of applications is severely degraded due to QASIOs through extensive investigation across the entire storage I/O stack of the Linux kernel encompassing virtual file system (VFS), page cache, Ext4 file system, JBD2 journaling layer, I/O scheduler, and block layer. Then, we classify the types of direct or indirect dependencies between file system operations and QASIOs. We also present how to detect each type of dependency in the Linux kernel. Finally, we devise a mechanism to dynamically prioritize QASIOs in the CFQ I/O scheduler, a de-facto I/O scheduler in the Linux kernel.

We have implemented and evaluated the proposed scheme on one of the latest Android-based smartphones, *Samsung Galaxy S5*. Our evaluation results with microbenchmarks show that the worst case latency of `creat()`, `fsync()`, and buffered `write()` is reduced by up to 98.4%, 87.1%, and 90.2%, respectively. In real workloads, the worst case launch time of the CONTACTS application is decreased by 44.8% under heavy asynchronous I/Os.

The rest of this paper is organized as follows. Section 2 explains some background to understand how the Linux kernel handles file I/O operations. Section 3 describes three problematic scenarios and Section 4 introduces QASIOs. The design and implementation details of how to detect QASIOs and boost them in the Linux kernel are presented in Section 5. Section 6 demonstrates evaluation results and Section 7 discusses the related work. Finally, Section 8 concludes the paper.

2 Background

2.1 I/O in the Android Platform

Android is one of the most widely used mobile platforms in the world. Android provides an application framework that allows a plenty of apps to operate simultaneously. An Android app consists of different type of components such as *Activity*, *Service*, *Broadcast Receiver*, and *Content Provider* which have an individual entry point and needs to be executed as a separate task. An app provides not only user interfaces, but also various background services performed by the request of the other components of the app or even by the other apps through *Intents* [2]. Recently, Android devices begin to support multi-window mode [16] beyond simple multi-tasking. The multi-window mode allows a user to divide one display screen into two and perform different tasks on each screen. For these reasons, a large number of tasks can run simultaneously at any moment in the Android system, yielding a large amount of I/Os at the same time.

2.2 Linux Kernel I/O Path

The Linux kernel is the core of the Android platform, which is responsible for managing system resources such as CPUs, memory, and various I/O devices. In particular, the storage I/O stack is one of the most complex parts in the Linux kernel as each file system operation is processed with the help of various layers such as virtual file system (VFS), Ext4 file system, page cache, JBD2 journaling layer, I/O scheduler, and block layer. In this subsection, we briefly describe a step-by-step procedure for processing a `write()` system call as shown in Figure 1. We choose the `write()` system call because it is slightly more complicated to handle compared to other system calls as it manipulates data and metadata at the same time. We assume that the default *ordered* journaling mode is used in the Ext4 file system.

1. Invoke a `write()` system call: A task passes the file descriptor, the address of the user data buffer, and the write size to the kernel through `write()`. This information is transmitted to the VFS layer. VFS updates necessary fields (such as timestamps and file size) of the file metadata (i.e., inode) by obtaining a JBD2 journal handle. The journal handle is obtained each time the metadata is modified to record the updated metadata in the journaling area. Then, VFS copies the requested data into the corresponding page in the page cache. The calling task returns from `write()` as soon as its data is copied into the page cache.

2. Make pages of file data dirty: The pages of file data written by VFS are marked as dirty. The Linux kernel accumulates these dirty pages up to a certain threshold.

3. Flush out dirty pages: If a dirty page stays for more than the expiration time or the amount of dirty pages

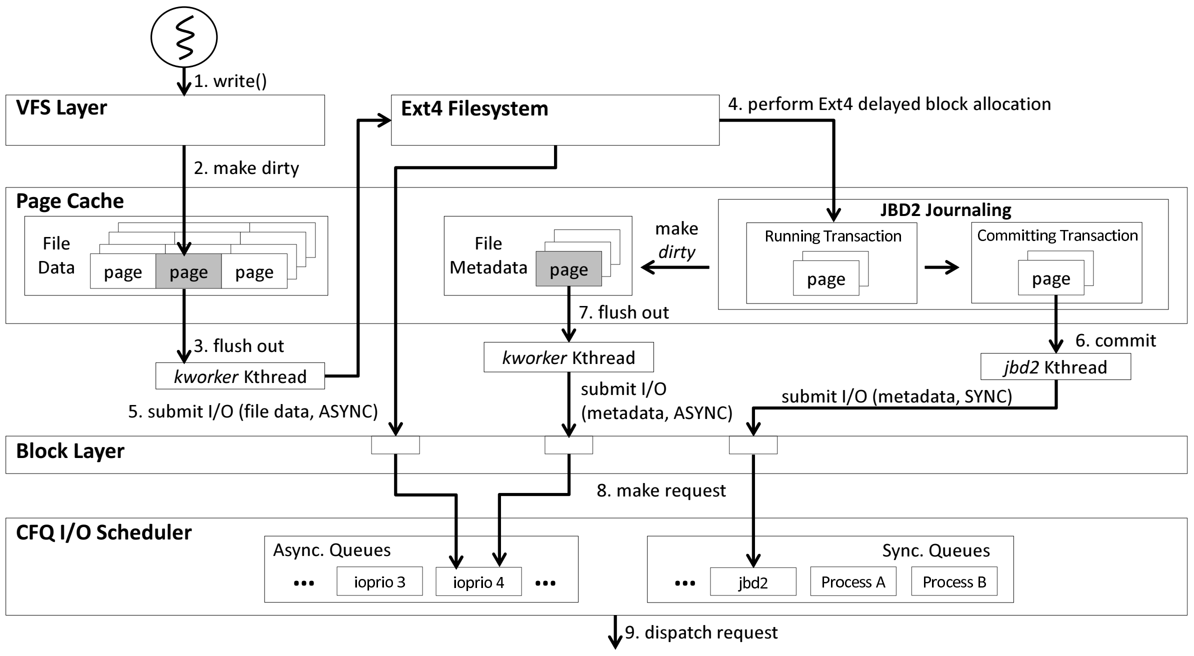


Figure 1: The processing of the `write()` system call in the Linux kernel

exceeds the background dirty ratio, they are forcibly flushed. The *kworker* kernel thread is executed periodically or synchronously in order to satisfy these requirements and asks the Ext4 file system to flush dirty pages if necessary.

4. Perform Ext4 delayed block allocation: By default, Ext4 performs block allocation when the file data is flushed from the page cache. During block allocation, Ext4 needs to modify file system metadata such as block bitmaps and group descriptors. Since these metadata should be also written into the journaling area, Ext4 obtains the journal handle of the running transaction and requests to the JBD2 module to manage them. Note that Ext4 associates a buffer head with each metadata page.

5. Submit dirty pages of file data: After block allocation, Ext4 submits dirty data pages to the block layer as *asynchronous* I/Os.

6. Commit JBD2 journaling: The running transaction is changed to the committing transaction after a predetermined time or a certain amount of buffer heads is gathered by the *jbd2* kernel thread. The metadata pages belonging to the committing transaction are submitted as *synchronous* I/Os by *jbd2*, because they must be written into the storage device rapidly for ensuring the file system integrity.

7. Flush out dirty metadata: After journal commit completes, the metadata pages included in the committing transaction are changed to the dirty state again, which enables checkpointing the metadata pages. Finally, *kworker* flushes out these metadata as *asynchronous* I/Os.

8. Make a request: Physically adjacent pages with the same I/O property among the submitted I/O requests are merged into one request through the block layer. These requests are forwarded to the I/O scheduler.

9. Dispatch a request: CFQ is the default I/O scheduler in the Linux kernel. CFQ has separate queues for synchronous I/Os and asynchronous I/Os. Synchronous I/O requests generated from a process is entered into the *synchronous* CFQ queue which is provided for each process. On the other hand, the *asynchronous* CFQ queue is shared by processes having the same I/O priority. Since most asynchronous I/Os are submitted by *kworker* (I/O priority 4) as shown in Figure 1, they are put into the same asynchronous CFQ queue. When dispatching a request, CFQ first selects a CFQ queue and then processes I/O requests in the selected queue in the order of logical sector number. Note that all the synchronous queues have higher priority than asynchronous queues in CFQ.

3 Problem and Motivation

This section presents three real-life scenarios as motivating examples which show reduced responsiveness under heavy asynchronous I/Os. The performance results of these scenarios are obtained from our test device (refer to Section 6 for details).

Scenario A: Launching the CONTACTS App:
The app start delay is a simple metric which shows the responsiveness of mobile devices. We observe that the

start time of the CONTACTS app gets slower and varies with a large standard deviation when a 4GB file is created in the background simultaneously. With 500 times of repeated tests, the worst case app start time increases by 140.0% over that of the normal case where there is no background I/O traffic.

Scenario B: Burst Mode in the CAMERA App:
The burst mode in the CAMERA app is a continuous high-speed shooting mode and is supported by many smartphones nowadays [19]. Our test device, *Samsung Galaxy S5*, shoots up to 30 shots by touching and holding on the shot icon. The next burst shot is possible shortly after the images taken by the first burst shot are processed. When the image size of each shot is large (8MB in the tested case), the intermittent delays occur between shots after the first burst shot. Finally, the burst shot performance is degraded by 19.0% than the ideal performance.

Scenario C: Installing the ANGRY BIRDS App:
The final scenario is to install the ANGRY BIRDS app downloaded from the Google Android market, when a 4GB file is written in the background. Since downloading the package file depends on the network performance, we measure the time to install the app from the package file pre-downloaded in the local storage. We observe that the average installation time increases by 35.0% when there are asynchronous I/Os in the background.

The underlying problem:
Many synchronous I/Os, such as `read()`'s or `write()`'s followed by `fsync()`, are issued during installing or launching an app. As mentioned in Section 2.2, the CFQ I/O scheduler gives higher priority to these synchronous I/Os over asynchronous I/Os. In spite of this, it is inevitable for the foreground task to experience some delay under heavy asynchronous I/Os for the following reasons. First, there can be a contention in holding a lock to modify the file system metadata. Second, it is possible that another asynchronous I/O is already in progress in the storage device when a synchronous I/O is dispatched. Third, there can be no room in memory or request queues for additional I/Os.

However, after investigating the three scenarios carefully, we find out that file system operations issued by the foreground task are significantly delayed by another reason. Although the specific condition is slightly different, the root cause of the problem is the same; the progress of a file system operation is blocked by *undispatched* asynchronous I/Os. This is an unexpected situation in the Linux kernel, resulting in an unpleasant consequence that the foreground task waits for the completion of asynchronous I/Os queued in the I/O scheduler. Since those asynchronous I/Os are not dispatched yet, the delay could be long as they may be served last by the I/O scheduler. In the next section, we show when this happens in more detail.

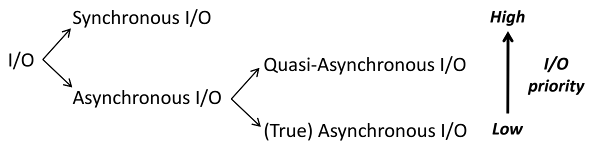


Figure 2: The classification of I/O requests

4 Quasi-Asynchronous I/O (QASIO)

This section defines a new type of I/O called quasi-asynchronous I/O (QASIO) and describes the types of dependencies on QASIOs. We also revisit the problematic scenarios shown in Section 3 to demonstrate how those scenarios are related to QASIOs.

4.1 Definition of Quasi-Asynchronous I/O

The Linux kernel traditionally categorizes I/O requests into the following two classes:

- *Synchronous I/O*: An I/O request is called synchronous when the calling task is blocked until the I/O request is completed. For this reason, the I/O scheduler such as CFQ treats synchronous I/Os in preference to asynchronous I/Os for better responsiveness. Typically, synchronous I/Os are created by `read()`, `fsync()`, and `sync()` system calls. However, `write()`'s can be made synchronous by opening a file with the `O_SYNC` flag. The *jbd2* kernel thread also generates synchronous I/Os when it commits journal data.
- *Asynchronous I/O*: Writing data to a file opened without the `O_SYNC` flag creates asynchronous I/Os. Asynchronous I/Os are flushed together by the *kworker* thread to maximize I/O throughput. They are handled in low priority in the I/O scheduler because no tasks waits for them. In this way, tasks can freely enjoy the benefits of the buffered I/O. Asynchronous I/Os are also produced when the file system metadata is written into the original location after journal commit.

In this paper, we introduce a new class of I/O called *quasi-asynchronous I/O (QASIO)* as depicted in Figure 2. A QASIO is defined as the I/O which is seemingly asynchronous but has the synchronous property because one or more tasks are waiting for its completion. This seems to be impossible in theory, but we show in the next subsection that it happens frequently in practice. Note that whether an I/O request is synchronous or asynchronous is determined when it is submitted to the block layer. In contrast, an existing asynchronous I/O is promoted to a QASIO *at run time* when a task gets blocked due to the asynchronous I/O. For better responsiveness, QASIOs should be given the higher priority than other (true) asynchronous I/Os.

4.2 Types of Dependencies on QASIO

Each task can have a *direct* or an *indirect* dependency on QASIOs. The direct dependency occurs when the execution of a task is blocked due to (quasi-) asynchronous I/Os. Figure 3 illustrates the situation where task A has a direct dependency on a QASIO. To identify when such a dependency exists, we have conducted an extensive analysis of the Linux kernel and the dynamic I/O patterns generated by file system calls. According to our analysis, we have identified the following four types of direct dependencies on QASIOs:

- *When modifying a metadata page* ($\mathbf{D}_{meta}$): This type of dependency can occur when a task invokes a file system call which modifies a metadata page (such as inodes, group descriptors, block bitmaps, inode bitmaps, and directory entries in Ext4). The target metadata page, made dirty by itself or the other tasks, may be already submitted as an asynchronous I/O by the *kworker* thread.
- *When modifying a data page* ($\mathbf{D}_{data}$): When a task appends data partially within a data page, it can be blocked since the target data page may be already flushed out asynchronously by the *kworker* thread. The task cannot proceed its execution until the data page hits the storage.
- *When guaranteeing data to be written back* ($\mathbf{D}_{sync}$): A task needs to wait for the completion of asynchronous I/Os when synchronizing or truncating the previously-issued file data in `fsync()` or `truncate()`. When performing `fsync()`, all the previous buffered writes are issued synchronously *as long as* they are still in the page cache. If calling `fsync()` is late or there are too many dirty pages in the page cache, some of them can be already flushed out as asynchronous I/Os. In this case, `fsync()` should wait until those asynchronously-issued I/Os are done.
- *When completing discard commands* ($\mathbf{D}_{discard}$): Currently, the *jbd2* kernel thread issues discard commands *asynchronously* for deallocated blocks, unlike other journal blocks which are issued synchronously. Hence, its execution is blocked on every journal commit until all the discard commands are completed. This delay in turn can affect the responsiveness of the foreground task (cf. $\mathbf{I}_{jcommit}$).

Sometimes, it is also possible that the execution of a task is being delayed due to another task that has a direct dependency on QASIOs. For example, Figure 3 shows

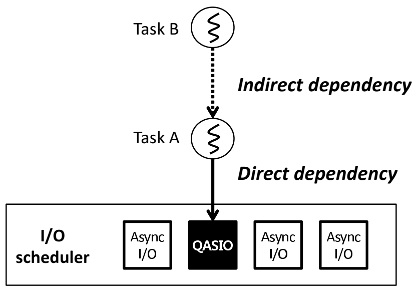


Figure 3: Direct and indirect dependency on QASIO

that task B is blocked because task A cannot make any progress due to the direct dependency on a QASIO. In this case, we call that task B has an indirect dependency on a QASIO. Typically, this situation arises when task A is blocked holding a resource that task B requires. Unlike the direct dependency, it is difficult to list all the possible types of indirect dependencies since the delay due to QASIOs can be propagated to other tasks in diverse and complicated ways. However, we found the following two types of indirect dependencies related to the JBD2 journaling which has a significant impact on the performance.

- *When unable to obtain a journal handle due to* $\mathbf{D}_{meta}$ *or* $\mathbf{D}_{data}$ ($\mathbf{I}_{jhandle}$): In Ext4, a task should obtain a journal handle to modify a metadata page or a data page. As mentioned before, the task can be blocked if the target page is already issued asynchronously, creating the $\mathbf{D}_{meta}$ or $\mathbf{D}_{data}$ type of dependency on QASIOs. Sooner or later, the transaction including the journal handle is started to be committed but the transaction is locked because the blocked task holds the journal handle. In this case, another task which attempts to perform any file operation is blocked since it fails to obtain a new journal handle.
- *When unable to complete* `fsync()` *due to* $\mathbf{D}_{discard}$ ($\mathbf{I}_{jcommit}$): This type of indirect dependency is observed only for the task that invokes `fsync()`. The `fsync()` system call needs to wait until the journal commit is completely done to ensure that the metadata of the corresponding file is written into the storage device. However, the processing time of the journal commit can be significantly prolonged since the *jbd2* kernel thread usually has a direct dependency of $\mathbf{D}_{discard}$ due to asynchronously-issued discard commands.

Whenever a foreground task interacting with a user has a direct or an indirect dependency on QASIOs, its execution has nondeterministic *hiccups* and the user can encounter sluggish responsiveness. Despite that there is room for additional I/Os in memory and request queues, the processing of system calls is blocked by the stacked

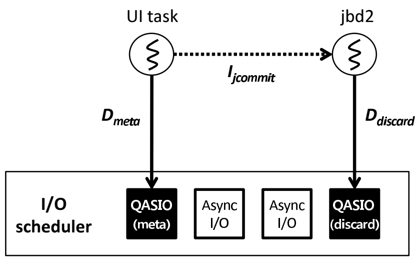


Figure 4: Dependencies on QASIOs in Scenario A

asynchronous I/Os in the request queue. More serious problem is that the lag due to QASIOs occurs not only in `fsync()`, but also such system calls as `creat()`, `chown()`, `unlink()`, and even *buffered* `write()`, where a user does not expect any delay. Depending on the storage performance, we observe that a single invocation of the `creat()` system call takes up to several seconds due to its dependency on QASIOs.

4.3 Revisiting Problematic Scenarios

We now take a closer look at the problematic scenarios in Section 3 and its relationship with QASIOs. Table 1 summarizes the major dependencies on QASIOs observed in each scenario.

4.3.1 Scenario A: Launching the CONTACTS App

When an app's UI shows up in the Android platform, several file system calls are made to update its states into databases (e.g., SQLite) or files (e.g., xml files) persistently. In our test device, launching the CONTACTS app is accompanied by a series of system calls such as `rename()`, `write()`, `fsync()`, and `unlink()`. These system calls all need to update the metadata. Therefore, they can have the $\mathbf{D}_{meta}$ type of dependency on QASIOs under bulky asynchronous I/Os, when they try to modify the metadata page which is being written back.

The UI task of the CONTACTS app has another indirect dependency of $\mathbf{I}_{jcommit}$ to QASIOs. At the end of each journal commit, the *jbd2* kernel thread has a direct dependency ($\mathbf{D}_{discard}$) due to the asynchronously-issued discard commands. If the journal commit is delayed due to $\mathbf{D}_{discard}$, the `fsync()` system call performed by the UI task is delayed as well since it cannot return until the metadata modification of the synchronized file is completely committed. Therefore, the UI task has two kinds of dependencies on QASIOs as depicted in Figure 4.

4.3.2 Scenario B: Burst Mode in the CAMERA App

After the first burst shot completes, several services are executed through the *Intents* transferred from the Android platform. One of them is the thumbnail maker task which generates thumbnails of the taken images. Since

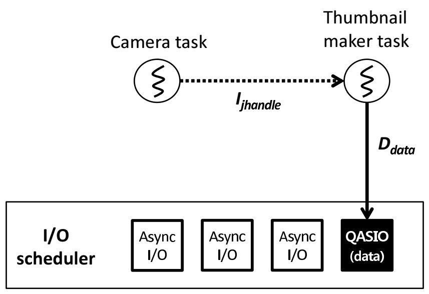


Figure 5: Dependencies on QASIOs in Scenario B

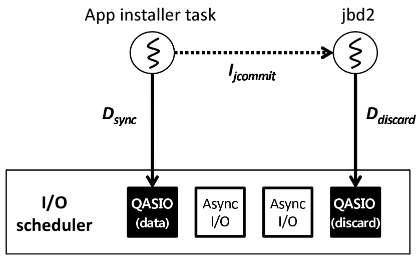


Figure 6: Dependencies on QASIOs in Scenario C

the thumbnail size is very small, the thumbnail maker task writes data in a small size (e.g., 1KB) repeatedly. This results in partial writes to the same data page, which can be blocked if the target data page is already being written back. Hence, the thumbnail maker task can have the $\mathbf{D}_{data}$ type of dependency on QASIOs.

Since the thumbnail maker is running in the background, it should not affect the responsiveness of the foreground CAMERA task. However, the problem is that the CAMERA task has an indirect dependency of $\mathbf{I}_{jhandle}$ to QASIOs via the thumbnail maker as shown in Figure 5. The thumbnail maker obtains a journal handle of the running transaction before copying the thumbnail image data to the data page, and falls into a sleep by the $\mathbf{D}_{data}$ dependency. After a certain period of time, the journal commit is started but the *jbd2* thread falls into the *locked* state since the thumbnail maker went asleep due to $\mathbf{D}_{data}$ with holding the journal handle of the transaction to be committed. When the CAMERA wants to acquire a journal handle to write additional images for the subsequent burst shots, it is eventually blocked. This is because the JBD2 journaling module does not give out any journal handle under the condition that the committing transaction is locked.

4.3.3 Scenario C: Installing the ANGRY BIRDS App

The dependencies on QASIOs arisen in this scenario are illustrated in Figure 6. The App installer task issues a number of *buffered* `write()` system calls in order to save the extracted data of the downloaded app to the app repository. To prevent data loss on sudden power failures, the App installer task invokes `fsync()` so that all the written data are flushed into the storage device. The `fsync()` system call is meant to write the data pages belonging to the corresponding file *synchronously*. However, when a large file is written in the background, the page cache is filled with dirty pages. In this case, the data pages to be `fsync()`'ed can be flushed out *asynchronously* by the *kworker* thread before the App installer invokes the `fsync()` system call. This leads to the $\mathbf{D}_{sync}$ type of dependency on QASIOs.

For the same reason as in scenario A, this scenario also has an indirect dependency of $\mathbf{I}_{jcommit}$ during `fsync()` due to asynchronously-issued discard commands.

5 Boosting QASIOs

In this section, we explain how to detect QASIOs efficiently at run time and boost them for better responsiveness during file system operations.

5.1 Design

When a task has a direct or an indirect dependency on a QASIO, its execution is blocked until the corresponding QASIO completes. This situation is somewhat similar to the priority inversion problem in scheduling where a high priority task cannot make any progress as a low priority task has a resource it requires. In this case, as the priority inheritance protocol does, the best way we can do to minimize the waiting time of the task is to give a higher priority to the QASIO and complete it quickly.

However, it is not easy in the current design of I/O schedulers since they do not know the presence of QASIOs and thus they have no idea of which one to boost. The various dependencies between tasks and QASIOs are formed *dynamically* at run time across various upper layers such as VFS, page cache, and Ext4 file system. This suggests that we have to have a run time mechanism which can detect QASIOs in the upper layers and notify the I/O scheduler to prioritize them.

The requirements for boosting QASIOs can be summarized as follows:

- *Req.(1): When a task is blocked waiting for the completion of an asynchronous I/O, the kernel should be able to give a hint about the existence of QASIO to the I/O scheduler.*
- *Req.(2): Upon the receipt of the hint from the kernel, the I/O scheduler should prioritize them among asynchronous I/Os.*

The *Req. (1)* is independent of the I/O scheduler used in the kernel, but *Req. (2)* needs to be re-implemented for each I/O scheduler. In this paper, we only show the implementation based on the CFQ I/O scheduler. However, the design can be easily applied to the other I/O schedulers. Figure 7 overviews our implementation for detecting and boosting QASIOs in the Linux kernel.

Table 1: The major dependencies on QASIOs in each scenario discussed in Section 4.3. (This table shows the major dependencies only. Sometimes other dependencies can occur as well. (B) represents that the dependency is associated with the background task, but the foreground task also has an indirect dependency on QASIOs.)

	Direct				Indirect	
Scenario	$\mathbf{D}_{meta}$	$\mathbf{D}_{data}$	$\mathbf{D}_{sync}$	$\mathbf{D}_{discard}$	$\mathbf{I}_{jhandle}$	$\mathbf{I}_{jcommit}$
A	✓			✓(B)		✓
B		✓(B)			✓	
C			✓	✓(B)		✓

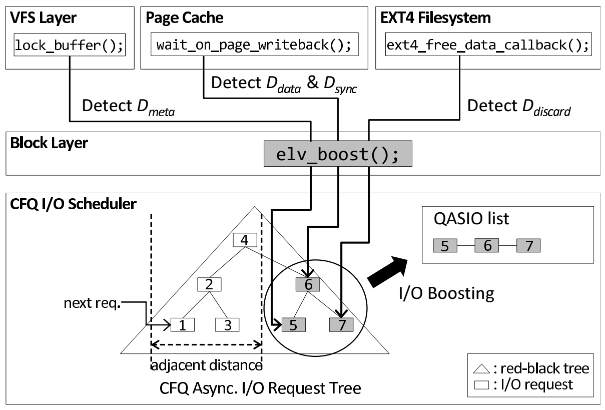


Figure 7: Implementation overview for detecting and boosting QASIOs in the Linux kernel

5.2 Detecting QASIOs

In this subsection, we present how QASIOs can be detected at run time in the Linux kernel. Since an indirect dependency on a QASIO occurs only when there is another direct dependency on the same QASIO, we only focus on detecting direct dependencies on QASIOs. If the direct dependency is resolved, the associated indirect dependency is terminated as well.

As we have seen in Section 4.2, there are four types of direct dependencies on QASIOs. Each can be detected as follows:

- *Detecting* $\mathbf{D}_{meta}$: In Ext4, a task accesses a metadata page through the associated buffer head structure. Before a task modifies a metadata page, it obtains an exclusive lock to the metadata page using the `lock_buffer()` kernel function. However, if the buffer head is already submitted to the block layer, the state of the buffer head is changed into the *locked* state and the execution of the task that attempts to acquire the same lock is suspended. Note that failing to obtain the lock does not necessarily mean that the target metadata page is submitted as an asynchronous I/O. Therefore, we have to double check whether the locked buffer head is being processed as an asynchronous I/O before making a decision.

- *Detecting* $\mathbf{D}_{data}$ *and* $\mathbf{D}_{sync}$: The dependency types of $\mathbf{D}_{data}$ and $\mathbf{D}_{sync}$ can be detected in the same location in case of Ext4. When a task wants to guarantee some data to be written back for synchronizing or truncating a file or to perform a partial write to a data page, it checks whether the previously-issued I/O has reached the storage device using the kernel function `wait_on_page_writeback()`. If necessary, the task waits in this function until the previous I/O is finished. Similar to the $\mathbf{D}_{meta}$ case, we should check whether the previous I/O is submitted as an asynchronous I/O.
- *Detecting* $\mathbf{D}_{discard}$: The kernel function `ext4_free_data_callback()` is registered to the JBD2 journaling module as a callback function. This function is called whenever the block deallocation is required at the end of journal commit. Unlike the above two cases, the *jbd2* kernel thread submits asynchronous discard operations directly in the callback function and then falls asleep waiting for the completion of those I/Os. Since it is obvious that *jbd2* generates QASIOs, no additional check is necessary.

To detect QASIOs in `lock_buffer()` and `wait_on_page_writeback()` functions, we should be able to quickly confirm whether the buffer head or the page, now being accessed, is issued as an asynchronous I/O. Since the Linux kernel has no way to represent this information, we added a special buffer head flag and modified the `submit_bh()` and `ext4_bio_write_page()` functions so that they set the flag when submitting asynchronous I/O requests. `submit_bh()` and `ext4_bio_write_page()` are used to submit a buffer head and a data page, respectively, to the underlying block layer. This special flag is unset after the I/O completes.

Algorithm 1 outlines how the actual detection of QASIOs is implemented in the function `wait_on_page_writeback()`. In the original implementation, a task simply waits in `wait_on_page_writeback()` until the writeback of the target page completes. Instead, we check whether the target page is submitted as an asynchronous I/O (lines 1–2) and if it is the case, we send the sector number of the detected QASIO to the I/O scheduler (lines 3–4). Note that even if the I/O scheduler is notified of the presence of a QASIO, it may fail to find it in the request queue (line 5) for the following two reasons.

Algorithm 1 A modified algorithm for `wait_on_page_writeback()` for detecting D_{data} and D_{sync}

```
 1: while the target page is already submitted do
 2:     if the page is issued as async. I/O then
 3:         extract the start sector number from the page
 4:         send the start sector number to I/O scheduler
 5:         if I/O scheduler fails to find the I/O then
 6:             set a timer of several clock ticks
 7:         end if
 8:     end if
 9:     wait until the page I/O completes or the timer expires
10: end while
```

First, the I/O request can be already dispatched to the storage device by the low-level device driver. In this case, we have no choice other than wait for the I/O completion. The second case is that the I/O request is staying temporarily in the *plug list* of the task, not in the request queue of the I/O scheduler. The plug list keeps I/O requests generated by a task for a short period time on the stack space of the task in order to increase the possibility of creating a larger request and to decrease a lock contention in the I/O scheduler [3]. Since the plug list is a private area that cannot be searched, we keep checking periodically until all the I/O requests in the plug list are flushed to the request queue. QASIOs can be detected similarly in `lock_buffer()` and `ext4_free_data_callback()` functions.

5.3 Prioritizing QASIO

Since QASIOs should be processed urgently, we give a higher priority to QASIOs than all the (true) asynchronous I/Os, but not more than any other synchronous I/Os. This is because we do not want that the boosting of QASIOs interferes with the responsiveness of synchronous I/Os. The actual implementation of handling QASIOs proceeds as follows.

Once a QASIO is detected, the information on the QASIO (i.e., start sector number) should be delivered to the I/O scheduler. For this purpose, we have added a new interface called `elv_boost()` in the elevator layer of the Linux kernel (cf. Figure 7). `elv_boost()` is an abstract interface which invokes a pre-registered function specific to each I/O scheduler, `cfq_boost_req()` in our case with the CFQ I/O scheduler.

For each asynchronous queue, we maintain a separate list of I/O requests called *QASIO list* as shown in Figure 7. In `cfq_boost_req()`, we traverse red-black trees of all the asynchronous queues and look for the I/O request which contains the received sector number of the QASIO. If it is found, an entry for the QASIO is inserted into the corresponding QASIO list.

In the original CFQ scheduler, whenever an asynchronous CFQ queue is selected for dispatching a request, CFQ finds the nearest request, specified by the `next_req` pointer, from the last processed request in the red-black tree and then sequentially dispatches I/O requests from that position. In our implementation, however, CFQ checks the QASIO list first and then dispatches QASIO requests if any. Moreover, if the QASIO list is empty, the current asynchronous queue yields the chance of I/O dispatching to another asynchronous queue which has a non-empty QASIO list. In this way, QASIOs are dispatched before any other asynchronous I/O requests.

6 Evaluation

This section presents the evaluation results with five microbenchmarks, three real-life scenarios, and two I/O benchmarks for Android.

6.1 Methodology

Our evaluation has been conducted on one of the latest smartphones, *Samsung Galaxy S5*, equipped with Exynos 5422 (including quad Cortex-A15 and quad Cortex-A7 ARM CPUs and Mali-T628 MP6 GPU), 2GB DRAM, and 16GB eMMC flash storage. It runs the Android platform version 4.4.2 (KitKat), with the Linux kernel 3.10.9. In Android, the dirty page expiration time is set to 2 seconds and the background dirty ratio to 5% by default.

In order to investigate the impact of QASIOs on the latency of various file system operations, we have used the following in-house microbenchmarks:

- M1: M1 iterates the creation of a 4KB file 500 times. In each iteration, M1 opens the same file with `creat()`, and then writes 4KB of data to the file using the buffered `write()`. Finally, it performs `fsync()` and closes the file with `close()`. Note that this microbenchmark mimics the storage I/O patterns of a database system such as SQLite [12].
- M2: M2 is the same as M1 except that the file size is increased to 1MB and the number of iterations is set to 200. The file data (1MB in size) is written using a single `write()` system call.
- M3: M3 creates a new file with `creat()` and repeats a 1KB-sized `write()` until the file size reaches 300MB.
- M4: In each iteration, M4 truncates the 2MB file created in the previous iteration to zero length using `truncate()`, recreates the same sized file using `write()`, and then closes the file with `close()`. This is repeated 500 times. The file for the first iteration is created manually before the execution.
- M5: M5 creates a single 4KB file by performing `creat()`, `write()`, `fsync()`, and `close()`, while another task truncates an existing 8GB file and writes 8GB of data again to the file.

We run all the microbenchmarks except M5 while a 8GB file is written in the background in order to generate asynchronous I/O operations. In addition to these microbenchmarks, the proposed scheme is evaluated with real-life scenarios discussed in Section 3 and two representative I/O benchmarks in Android, Antutu and RL-Bench.

6.2 Microbenchmarks

Figure 8 compares the total elapsed time of each microbenchmark according to the type of dependency boosted. The results are normalized to NONE in which no special handling is performed for QASIOs. Each $\mathbf{D}_{data}$+$\mathbf{D}_{sync}$, $\mathbf{D}_{meta}$, and $\mathbf{D}_{discard}$ represents the case where only the specified type of dependency is detected and boosted. Note that $\mathbf{D}_{data}$ and $\mathbf{D}_{sync}$ types cannot be boosted separately, as they are detected in the same location (cf. Section 5.2). Finally, ALL means that all kinds of optimizations are applied for QASIOs. Table 2 presents the latency of key file system operations before and after applying the optimizations for QASIOs. Overall, we can see that boosting QASIOs improves the total elapsed time by up to 83.1%. The proposed scheme also reduces the worst case latency of each file system operation by up to 98.4%. The detailed analysis on the result of each microbenchmark is as follows.

In M1, when all the dependency types are boosted, the total elapsed time is reduced by 83.1% as the average latency of `creat()` and `fsync()` is improved by 99.1% and 63.7%, respectively. In each iteration of M1, `creat()` modifies metadata pages and also incurs discard operations as it creates the file with the same name. Hence, `creat()` and `fsync()` will have the $\mathbf{D}_{meta}$ and $\mathbf{I}_{jcommit}$ dependency, respectively. This is why the most of reduction in the total elapsed time comes when $\mathbf{D}_{meta}$ and $\mathbf{D}_{discard}$ types are boosted.

The M2's results show the similar trend as M1 except that boosting $\mathbf{D}_{data}$+$\mathbf{D}_{sync}$ is as effective as $\mathbf{D}_{meta}$ or $\mathbf{D}_{discard}$. As the file size becomes larger, it is likely that *kworker* flushes out some of data pages asynchronously before `fsync()` is called. Consequently, unlike in M1, `fsync()` will have the $\mathbf{D}_{sync}$ dependency in M2.

In M3, the most dominant operation affecting the overall performance is the *buffered* `write()`. From Table 2, we can observe that the latency of `write()` is reduced by 47.4% on average and by 90.2% in the worst case. `write()` suffers from the $\mathbf{D}_{data}$ dependency since 1KB of data is written into a data page partially. Therefore, the most of performance improvement is achieved by boosting the $\mathbf{D}_{data}$ type of dependency as Figure 8 illustrates.

In M4, the key file system operation affected by QASIOs is `truncate()`. If the file data is already issued

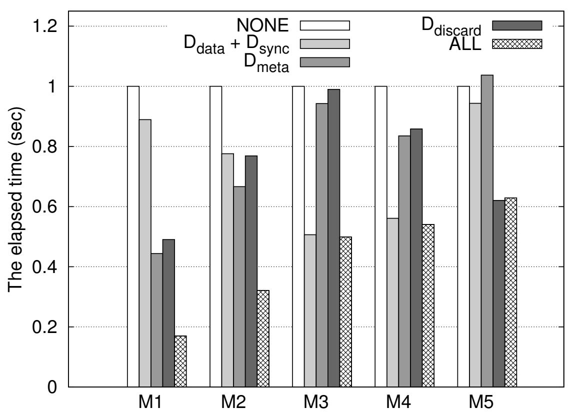


Figure 8: The normalized total elapsed time of each microbenchmark according to the dependency type boosted

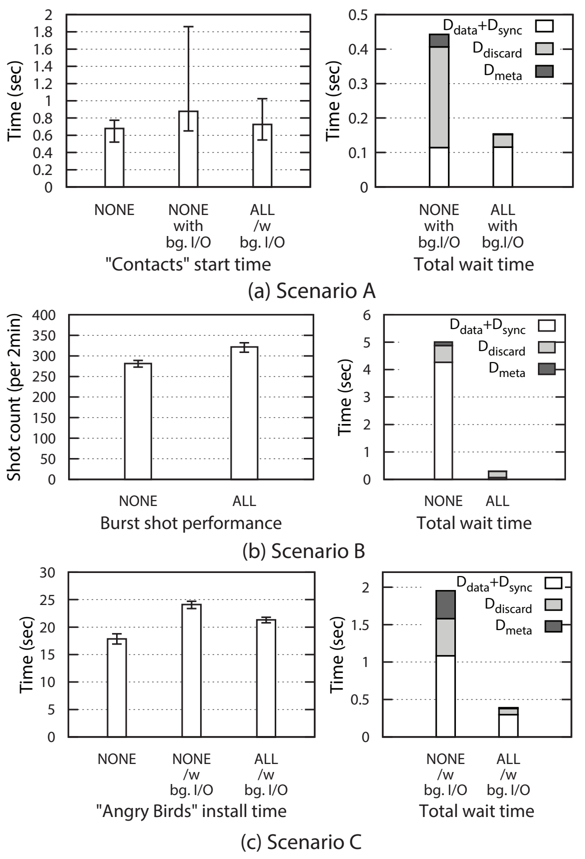


Figure 9: Results of three real-life scenarios

asynchronously, `truncate()` should wait for the completion of those asynchronous I/Os. Hence, boosting the $\mathbf{D}_{sync}$ type of dependency is most helpful for M4.

In case of M5, *jbd2* has the $\mathbf{D}_{discard}$ type of dependency on asynchronously-issued discard commands as another task truncates a large file. In this case, calling `fsync()` to synchronize just 4KB of data takes 13.27 seconds on average due to the $\mathbf{I}_{jcommit}$ dependency. However, when we boost the $\mathbf{D}_{discard}$ type, the latency is decreased to 6.85 seconds.

Since QASIOs are prioritized over other asynchronous I/Os in the I/O scheduler, boosting QASIOs can have a negative impact on the throughput of asynchronous I/Os. To investigate this effect, we have measured the throughput of creating a 8GB file in the background while performing the M1 microbenchmark. According to our measurement results, the throughput is decreased by 15.4%, from 46.2MB/s to 39.1MB/s. We believe this is acceptable considering that the total elapsed time of the foreground task in M1 is improved by 83.1%.

6.3 Real-life Scenarios

Figure 9 depicts the impact of boosting QASIOs in three real-life scenarios described in Section 3. On the right side of Figure 9, we also show the total time spent on waiting for the completion of QASIOs. These times are measured in the kernel functions described in Section 5.2 where each type of dependency is detected. In Figure 9(a) and (c), the phrase "with bg. I/O" indicates the case where a 4GB file is created in the background simultaneously in order to generate asynchronous I/Os.

In Scenario A, we have measured the time to launch the CONTACTS app. Under heavy asynchronous I/Os, the launch time is increased by 29.4% on average. In the worst case, the app start is slowed down by a factor of 2.4. However, boosting QASIOs effectively reduces the worst case launch time by 44.8%. Similar to the M1 microbenchmark, the most of improvement comes from boosting $\mathbf{D}_{meta}$ and $\mathbf{D}_{discard}$, as the CONTACTS task has the $\mathbf{D}_{meta}$ and $\mathbf{I}_{jcommit}$ dependency on QASIOs. The total wait time on the corresponding kernel function of $\mathbf{D}_{meta}$ and $\mathbf{D}_{discard}$ has been reduced by 96.1% and 87.4%, respectively.

In Scenario B, we can see that the shot count in the burst mode is improved by 14.4% on average when we boost QASIOs. The thumbnail maker has the $\mathbf{D}_{data}$ dependency on QASIOs, however the total wait time due to $\mathbf{D}_{data}$ is reduced by 98.4%. As the direct dependency between the thumbnail maker and QASIOs is resolved quickly, the burst mode performance of the CAMERA app has been improved as well.

Finally, the average installation time of the ANGRY BIRDS app is slowed down by 35.0% under the heavy asynchronous I/Os in the background. However, we observe that the average installation time is improved by 11.5% through the boosting of QASIOs. As mentioned in Section 4.3.3, the ANGRY BIRDS app has the $\mathbf{D}_{sync}$ and $\mathbf{I}_{jcommit}$ dependency on QASIOs. Accordingly, the most of reduction in the installation time comes from boosting the $\mathbf{D}_{sync}$ type. Boosting $\mathbf{D}_{discard}$ and $\mathbf{D}_{meta}$ also contributes to reducing the installation time since some

Table 2: The latency of key file system operations in each microbenchmark. (The time unit is millisecond for M1 – M4, while it is second for M5)

Opt	M1				M2				M3		M4		M5	
	`creat()`		`fsync()`		`creat()`		`fsync()`		`write()`		`truncate()`		`fsync()`	
	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max
NONE	119.57	1435.54	98.24	1119.62	109.01	1397.64	172.05	1417.82	0.19	1813.54	62.60	1632.15	13.27	13.87
ALL	**1.02**	**39.15**	**35.64**	**144.69**	**3.90**	**22.52**	**69.24**	**298.83**	**0.10**	**177.40**	**12.85**	**334.57**	**6.85**	**7.11**

metadata modifications and discard operations are performed during the app install procedure.

6.4 I/O Benchmarks for Android

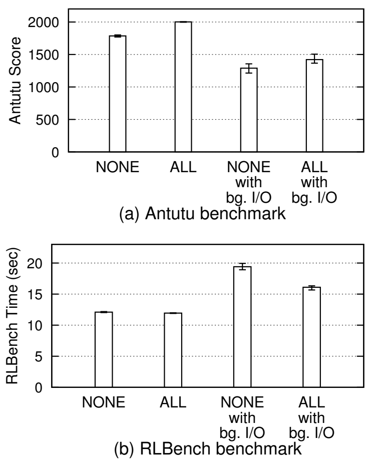


Figure 10: Results of Antutu and RLBench

Finally, we have evaluated the proposed scheme with two famous I/O benchmarks for Android, Antutu [6] and RLBench [15]. Antutu is a comprehensive Android benchmark which provides several modes for measuring the performance of CPU, memory, storage system, etc. In this evaluation, we use the *Database IO* mode that estimates the storage I/O performance with database workloads. The Antutu benchmark reports the final score as a result of the performance measurement, where the higher score means the better performance.

RLBench is a benchmark for measuring the performance of Android devices under SQLite workloads. Since it makes SQLite generate storage I/O intensive workloads, the RLBench's result is closely related to the storage I/O performance in general. Unlike Antutu, RLBench shows the elapsed time to process the predefined set of SQL queries, hence the lower time means the better performance.

The results of Antutu and RLBench are displayed in Figure 10. We run each benchmark with (labeled as "with bg. I/O") and without asynchronous I/O operations. The asynchronous I/Os are generated by writing a 4GB file in the background while running the benchmarks.

In Antutu, the base score is measured at 1,785 when there is no asynchronous I/O operations in the background. Due to asynchronous I/Os, the score is decreased to 1,289. However, the proposed scheme improves the score to 1,421 which is smaller than the base score by 20.4%. An interesting observation is that boosting QASIOs improves the performance of Antutu by about 12.0% even when there is no asynchronous I/Os in the background. This means Antutu itself issues asynchronous I/Os and its performance is also affected by QASIOs.

The result of RLBench also shows that the proposed scheme successfully improves the storage I/O performance. When there are asynchronous I/O operations in the background, the elapsed time is reduced by 17.1% by boosting QASIOs.

7 Related Work

Mobile devices usually employ NAND flash memory as the media of the main storage system. Kim et al. show that the storage performance is a limiting factor for the performance of several common applications for mobile devices through extensive experiments with various flash storage systems [9]. Since NAND flash memory shows very different characteristics compared to hard disk drives, prior work attempts to revisit various operating system mechanisms and policies which have been optimized for rotating media. As a result of these efforts, several file systems [4, 13] and I/O schedulers [14, 17] have been proposed which are optimized for the characteristics of flash storage.

Recently, many researches have focused on optimizing the I/O stack of the Linux kernel in accordance with the I/O characteristics of SQLite, a lightweight transactional database engine provided by the Android platform. Since most applications heavily utilize SQLite to keep their application-specific data persistently, the overall performance of mobile applications is known to highly depend on the SQLite's performance. In particular, Lee et al. have observed that running SQLite on top of the Ext4 file system produces very inefficient I/O patterns to the storage system [12]. Based on the observation, Jeong et al. propose the elimination of unnecessary metadata journaling, external journaling, and a polling-based I/O mechanism to improve the journaling efficiency [8]. Similarly, Shen et al. propose the enhanced journaling mechanism for the Ext4 file system in the SQLite environment to solve the so-called *journaling of journal* problem [18].

In this paper, we focus on the fact that the responsiveness of file system operations is severely degraded when the system has lots of storage I/O operations asynchronously issued. This is an inherent problem in handling I/O requests, being independent of file systems and I/O schedulers. Therefore, our approach is largely orthogonal to previous researches. Note that the proposed scheme reduces the latency of `fsync()` successfully, which is known to be performed frequently by SQLite. Therefore, boosting QASIOs will be also helpful in improving the performance of SQLite as exemplified in the result of RLBench.

The recent Linux kernel allows to update data pages while they are under writeback by disabling the *stable pages* feature, which successfully eliminates any $\mathbf{D}_{data}$ dependency [5]. However, this can be unacceptable in future mobile devices since the hardware-supported encryption during I/O is seriously being considered. Also, the $\mathbf{D}_{discard}$ dependency can be removed if a userspace program issues a *fstrim* command in a batch manner at convenient times when the device is idle, as introduced in the recent Android platform [1]. However, this may not be a complete solution since the I/O performance of the underlying flash storage can be suddenly degraded if discard commands are not issued at a proper time.

8 Conclusions

This paper introduces a new type of I/O called Quasi-Asynchronous I/O (QASIO). The QASIO is the I/O operation which is seemingly asynchronous but has the synchronous property since one or more tasks are blocked until the I/O operation is completed. As the system handles asynchronous I/Os in the perspective of maximizing throughput not latency, the responsiveness of the blocked tasks is significantly degraded. In particular, in mobile devices where most applications interact with users all the time, the quality of user experiences suffers from QASIOs.

In order to address this problem, we propose a novel scheme to detect QASIOs and boost them in the Linux kernel. We have implemented and evaluated the proposed scheme on the latest Android-based smartphone, *Samsung Galaxy S5*. By performing various microbenchmarks, real-life scenarios, and Android I/O benchmarks, we confirm that boosting QASIOs is effective in improving the responsiveness of file system operations. We plan to analyze the effect of boosting QASIOs on more diverse platforms including servers and low-end smartphones.

9 Acknowledgements

We would like to thank the anonymous reviewers and our shepherd, Youjip Won, for their valuable comments. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea Government (MSIP) (No. 2013R1A2A1A01016441).

References

[1] Android 4.3 update brings trim to all nexus devices. `http://www.anandtech.com/show/7185/android-43-update-brings-trim-to-all-nexus-devices`.

[2] Android developers. `http://developer.android.com`.

[3] Explicit block device plugging. `http://lwn.net/Articles/438256`.

[4] F2fs: Introduce flash-friendly file system. `https://lwn.net/Articles/518718/`.

[5] Optimizing stable pages. `http://lwn.net/Articles/528031`.

[6] ANTUTU LABS. AnTuTu Benchmark. `https://play.google.com/store/apps/details?id=com.antutu.ABenchMark5`.

[7] GARTNER, INC. Gartner says worldwide traditional PC, tablet, ultramobile and mobile phone shipments are on pace to grow 6.9 percent in 2014. `http://www.gartner.com/newsroom/id/2692318`.

[8] JEONG, S., LEE, K., LEE, S., SON, S., AND WON, Y. I/O stack optimization for smartphones. In *USENIX ATC* (2013), pp. 309–320.

[9] KIM, H., AGRAWAL, N., AND UNGUREANU, C. Revisiting storage for smartphones. In *FAST* (2012), pp. 1–14.

[10] KIM, H., AND SHIN, D. Optimizing storage performance of android smartphone. In *ICUIMC* (2013).

[11] KIM, W.-H., NAM, B., PARK, D., AND WON, Y. Resolving journaling of journal anomaly in android i/o: multi-version b-tree with lazy split. In *FAST*, pp. 273–285.

[12] LEE, K., AND WON, Y. Smart layers and dumb result: IO characterization of an android-based smartphone. In *EMSOFT* (2012), pp. 23–32.

[13] LU, Y., SHU, J., AND WANG, W. ReconFS: a reconstructable file system on flash storage. In *FAST* (2014), pp. 75–88.

[14] PARK, S., AND SHEN, K. FIOS: a fair, efficient flash I/O scheduler. In *FAST* (2012), pp. 1–15.

[15] REDLICENSE LABS. RL Benchmark: SQLite. `https://market.android.com/details?id=com.redlicense.benchmark.sqlite`.

[16] SAMSUNG. How do i use multi window mode (multitasking) on my Samsung Galaxy Note II? `http://www.samsung.com/us/support/howtoguide/N0000004/8829/62396`.

[17] SHEN, K., AND PARK, S. FlashFQ: A fair queueing I/O scheduler for flash-based SSDs. In *USENIX ATC* (2013), pp. 67–78.

[18] SHEN, K., PARK, S., AND ZHU, M. Journaling of journal is (almost) free. In *FAST* (2014), pp. 287–293.

[19] WIKIPEDIA. Burst mode (photography). `http://en.wikipedia.org/wiki/Burst_mode_(photography)`.

Failure-Atomic Updates of Application Data in a Linux File System

Rajat Verma[1] *Anton Ajay Mendez*[1] *Stan Park*[2]
Sandya Mannarswamy[1] *Terence Kelly*[2] *Charles B. Morrey III*[2]

[1]*Hewlett-Packard Storage Division* [2]*Hewlett-Packard Laboratories*

Abstract

We present the design, implementation, and evaluation of a file system mechanism that protects the integrity of application data from failures such as process crashes, kernel panics, and power outages. A simple interface offers applications a guarantee that the application data in a file always reflects the most recent successful `fsync` or `msync` operation on the file. Our file system furthermore offers a new `syncv` mechanism that failure-atomically commits changes to *multiple* files. Failure-injection tests verify that our file system protects the integrity of application data from crashes and performance measurements confirm that our implementation is efficient. Our file system runs on conventional hardware and unmodified Linux kernels and will be released commercially. We believe that our mechanism is implementable in any file system that supports per-file writable snapshots.

1 Introduction

Many applications modify data on durable media, and failures during updates—application process crashes, OS kernel panics, and power outages—jeopardize the integrity of the application data. We therefore require solutions to the fundamental problem of *consistent modification of application durable data* (CMADD), i.e., the problem of evolving durable application data without fear that failure will preclude recovery to a consistent state.

Existing mechanisms provide imperfect support for solving the CMADD problem. Relational databases offer ACID transactions; similarly, many key-value stores allow failure-atomic bundling of updates [2, 13, 14]. Despite the obvious attractions of transactions, both kinds of databases can lead to two difficulties: First, in-memory data structures do not always translate conveniently or efficiently to and from database formats; repeated attempts to smooth over the "impedance mismatch" between data formats have met with limited success [19]. Second, the complexity of modern databases offers fertile ground for implementation bugs that negate the promise of ACID: A recent study has shown that widely used key-value and relational databases exhibit erroneous behavior under power failures; the proprietary commercial databases tested *lose data* [36].

File systems strive to protect their *internal metadata* from corruption, but most offer no corresponding protection for *application* data, providing neither transactions on application data nor any other unified solution to the CMADD problem. Instead, file systems offer primitives for controlling the order in which application data attains durability; applications shoulder the burden of restoring consistency to their data following failures. Added to the inconvenience and expense of implementing correct recovery is the inefficiency of the sequences of primitive operations required for complex updates: Consider, for example, the chore of failure-atomically updating a set of files scattered throughout a POSIX-like file system. Remarkably, the vast majority of file systems do not provide the straightforward operation that CMADD demands: the ability to modify application data in (sets of) files failure-atomically and efficiently.

We present the design, implementation, and evaluation of failure-atomic application data updates in HP's Advanced File System (AdvFS), a modern industrial-strength Linux file system derived from DEC's Tru64 file system [1]. AdvFS provides a simple interface that generalizes failure-atomic variants of `writev` [8] and `msync` [20]: If a file is opened with a new `O_ATOMIC` flag, the state of its application data will always reflect the most recent successful `msync`, `fsync`, or `fdatasync`. AdvFS furthermore includes a new `syncv` operation that combines updates to multiple files into a failure-atomic bundle, comparable to the multi-file transaction support in Windows Vista TxF [17] and TxOS [22] but much simpler than the former and more capable than the latter. The size of transactional updates in AdvFS is limited only by the free space in the file system. AdvFS requires no special hardware and runs on unmodified Linux kernels.

The remainder of this paper is organized as follows: Section 2 situates our contributions in the context of prior work. Section 3 describes AdvFS and the features that made it possible to implement failure-atomic updates of application data. Section 4 presents experimental evaluations of both the correctness and performance of AdvFS, and Section 5 concludes with a discussion.

2 Related Work

Most widely deployed mainstream file systems offer only limited and indirect support for consistent modification of application durable data (CMADD).[1] Semantically weak OS interfaces are partly to blame. For example, POSIX permits `write` to succeed *partially*, making it difficult to define atomic semantics for this call [30]. Synchronization calls such as `fsync` and `msync` constrain the order in which application data reaches durable media, and recent research has proposed decoupling ordering from durability [5]. However applications remain responsible for building CMADD solutions (e.g., atomicity mechanisms) atop ordering primitives and for reconstructing a consistent state of application data following a crash. Experience has shown that custom recovery code is difficult to write and prone to bugs. Sometimes applications circumvent the need for recovery by using the one failure-atomicity mechanism provided in conventional file systems: file rename [11]. For example, desktop applications can open a temporary file, write the entire modified contents of a file to it, then use `rename` (or a specialized equivalent [7, 23]) to implement an atomic file update—a reasonable expedient for small files but untenable for large ones.

FusionIO provides an elegant and efficient mechanism for solving the CMADD problem for data on their flash-based storage devices: a failure-atomic `writev` supported by the flash translation layer [8]. The MySQL database exploits this new mechanism to eliminate application-level double writes and thereby improve performance substantially [29]. Still more impressive gains are available to applications architected from scratch around the new mechanism. For example, a key-value store designed to exploit the new feature achieves both performance and flash endurance benefits [15]. The limitations of failure-atomic `writev` are that it requires special hardware, applies only to single-file updates, and does not address modifications to memory-mapped files.

Fully general support for failure-atomic bundles of file modifications is surprisingly rare. Windows Vista TxF supports such a capability, but the feature is deprecated because its formidably complex interface has impaired adoption [17]. TxOS includes a simpler interface to the same capability, but with a limitation: Because TxOS implements atomic file updates via the file system journal, transaction size is limited by the size of the journal [22]. Valor implements in the Linux kernel a transactional file update API with seven new system calls that support inter-process isolation even in the presence of non-transactional accesses by legacy applications [25]. The price that transaction-aware applications pay for this sophisticated support includes a substantial burden of logging: Applications must perform a `Log Append` syscall prior to modifying a page of a file within a transaction, which is awkward at best for the important case of random STOREs to a memory-mapped file.

An attractive approach to the CMADD problem on emerging durable media is a persistent heap supporting atomic updates via a transactional memory (TM) interface. Mnemosyne [31] and Hathi [24] implement such mechanisms for byte-addressable non-volatile memory (NVM) and flash storage, respectively. Persistent heaps obviate the need for separate in-memory and durable data formats: Applications simply manipulate in-memory data structures using LOAD and STORE instructions, which seems especially natural for byte-addressable NVM. One limitation of these systems is that they do not support conventional file operations; another is that they are tailored to specific durable media. Finally, they employ software TM, which carries substantial overheads.

Persistent heaps can be implemented for conventional block storage and need not employ TM. Recent examples include Software Persistent Memory (SoftPM) [9] and Ken [34], whose persistent heaps expose `malloc`-like interfaces and support atomic checkpointing. Such approaches provide ergonomic benefits and are compatible with conventional hardware, but their atomic-update mechanisms entail substantial complexity and overheads. For example, SoftPM automatically copies volatile data into persistent containers as necessary through a novel hybrid of static and dynamic pointer analysis, making development easier and less error-prone. However SoftPM tracks data modification in coarse-grained chunks of 512 KB or larger, which can lead to write amplification at the storage layer. Ken's user-space persistent heap tracks modifications at 4 KB memory-page granularity, which may reduce write amplification, but Ken writes each modified page to storage twice (to a REDO log synchronously and in-place asynchronously).

Failure-atomic `msync` ensures that application data in the backing file always reflects the most recent successful `msync` call [20]. It is easy to layer increasingly sophisticated higher-level abstractions atop this foundation, e.g., persistent heaps which in turn can slide beneath general-purpose libraries of data structures and algorithms such as C++ STL. Although it supports the style of programming natural to non-volatile memory,

[1] Spillane et al. provide an extensive review of research literature on transactional file systems [25].

failure-atomic msync can be implemented on conventional block storage. A kernel-based implementation of failure-atomic msync suffers at least three shortcomings: The need to run a modified kernel impedes adoption, the use of the file system journal limits transaction sizes, and data modifications are written to storage twice (once in the journal and once in-place) [20]. A user-space implementation of a similar mechanism eliminates the first two problems and has been deployed in commercial production systems [3], but it does not support fully general file manipulations and it retains the double write due to logging.

AdvFS solves the CMADD problem directly and combines many of the advantages of prior approaches. The interface is both simple and general: Opening a file with a new O_ATOMIC flag guarantees that the file's application data will reflect the most recent synchronization operation, regardless of whether the file was modified with the write or mmap families of interfaces (or both). Our new syncv operation ensures that updates to a *set* of files are atomic. Because it includes failure-atomic msync as a special case, AdvFS offers the same advantages as a foundation atop which persistent heaps and other abstractions may be layered. AdvFS does not rely on the file system journal to implement atomic updates; it avoids double writes and the size of atomic updates is limited only by the amount of free space in the file system. Adopting AdvFS is relatively easy because it runs on standard Linux kernels and requires no special hardware. Its atomic-update interface admits implementation atop both conventional block storage as well as emerging byte-addressable NVM, and thus it provides a smooth transition path from the former to the latter. Finally, AdvFS for Linux is not a research prototype. It is an extensively modernized production-quality upgrade and Linux port of DEC's Tru64 file system, and it is scheduled for commercial release in March 2015 as part of HP Storage appliances.

As described in Section 3, implementing O_ATOMIC is straightforward in file systems that support per-file writable snapshots [4, 12, 28, 32]. We believe that most could implement O_ATOMIC and syncv without prohibitive cost or complexity, which in turn would make it much easier to write robust applications.

3 Implementation

AdvFS is a modern, update-in-place, journaling Linux file system developed internally for commercial storage appliances, designed to be scalable and performant for multiple use cases and workloads. AdvFS for Linux evolved from DEC's Tru64 file system, which was open sourced in 2008 [1, 10] and has been rewritten extensively for modern storage devices, with enterprise scalability and reliability. It supports a number of advanced capabilities such as the ability to add/remove storage devices online, support multiple file systems on the same storage pool, and take *clones* (described below) and snapshots at file, directory, and FS level.

Like other modern file systems that support storage pools [35], AdvFS decouples the logical file hierarchy from the physical storage. The logical file hierarchy layer implements the naming scheme and POSIX-compliant functions such as creating, opening, reading, and writing files. The physical storage layer implements write-ahead logging, caching, file storage allocation, file migration, and physical disk I/O functions. AdvFS is comparable in performance and feature richness to most modern Linux file systems; due to space constraints we omit a detailed description of AdvFS and comparisons with other modern FSes. We designed and implemented O_ATOMIC on AdvFS and exposed it to applications through the conceptually simple and familiar interface of open followed by write/fsync or mmap/msync.

O_ATOMIC leverages a *file clone* feature developed to support use cases such as virtual machine cloning. A file clone is a writable snapshot of the file. AdvFS implements file cloning utilizing a variant of copy-on-write (COW) [21], illustrated in Figure 1. When a file is cloned, a copy of the file's inode is made. The inode includes the file's block map, a data structure that maps logical file offsets to block numbers on the underlying block device. Since the original file and its clone have identical copies of the block map, they initially share the same storage. When a shared block is eventually written to, either in the original file or in its clone, a copy of the block is made and remapped to a different location on the block device. Since COW results in new data blocks being assigned to the original file, it has the downside that it can fragment the original file; AdvFS supports online defragmentation, which can mitigate this difficulty. Efficient clone implementation in AdvFS enabled a simple but effective implementation of O_ATOMIC.

When a file is opened with O_ATOMIC, a clone of the file is made (Figure 1(a)-(b)). This clone is not visible in the user visible namespace but exists in a hidden namespace accessible by AdvFS. When the file is modified the changed blocks are remapped via COW (Figure 1(c)). The clone still points to the blocks of the file at the time the file was opened. On a subsequent call to fsync/msync the existing clone is deleted and a new one is created to track the latest version of the file (Fig-

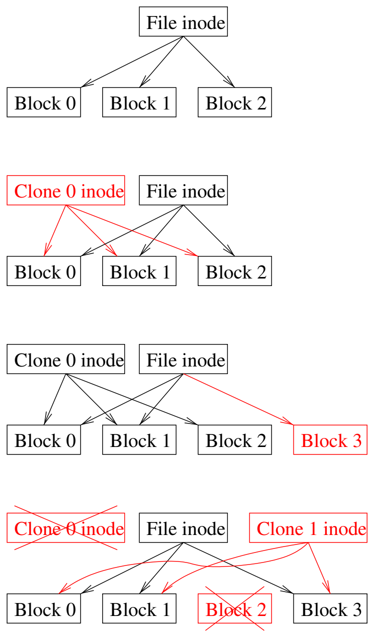


Figure 1: File clones implement atomic updates.

ure 1(d)). On the close of a file opened with `O_ATOMIC`, the original file is replaced with the clone.

If the system crashes, recovery of an `O_ATOMIC` file is delayed until the file is accessed again. The file system's path name lookup function checks if the file has a clone in the hidden name space; this is a very inexpensive check that is performed on *every* file open. If a clone exists, it is renamed to the user visible file and a handle to it is returned (we require *writable* clones rather than read-only snapshots because of this scenario). AdvFS's "lazy," per-file recovery offers several attractions: Consider, for example, a kernel panic that occurs while many processes are atomically updating many files. Upon reboot, the file system will recover quickly because the in-progress updates, interrupted by the crash, trigger no recovery actions when the file system is mounted. The net effect is that applications that do not need recovery from interrupted atomic updates (e.g., applications that are merely *reading* files) do not share the recovery-time penalty incurred by the crash; only those applications that *benefit* from application-consistent recovery pay the penalty. Lazy recovery could in principle lead to the accumulation of hidden clones; it would be straightforward to delete clones in the background.

While our support for atomic file durability for `O_ATOMIC` is built atop the clone feature of AdvFS, alternative implementations are also possible, such as using delayed journal writeback [20]. Using journal writeback to achieve atomic durability has the disadvantage that the size of a single-file atomic update is limited by the size of the journal; another downside is that journaling can lead to double writes of modified data. Our approach does not have these limitations but requires that the file system provide the ability to create per-file clones. In our experience, implementing `O_ATOMIC` atop a per-file cloning capability is relatively straightforward, and we believe that similar implementations on other file systems that support cloning are possible. As of this writing several open source and commercial file systems support per-file clones [4, 12, 28, 32].

Applications that use the `O_ATOMIC` feature of AdvFS must obey a few simple rules. The only new rule is that overlapping or concurrent modifications to a single file via multiple file descriptors void the failure-atomicity guarantee and should be avoided. Due to various subtleties it is not possible to define unambiguous semantics in such cases, so different processes/applications must coordinate their access to files. The remaining rules are not specific to AdvFS or `O_ATOMIC`: As in other file systems, multi-threaded concurrent accesses to a single file must be "orderly," in the sense that data races, atomicity violations, and other concurrency bugs must be avoided. As in other file systems, programmers must ensure that data being committed via `fsync` or `msync` is not being modified by other threads during those calls.

3.1 Multi-File Atomic Updates: `syncv`

In many situations it is necessary to failure-atomically update *several* files. For example, the popular SQLite database management system stores separate databases in separate files, and it supports transactions that atomically update multiple databases. To failure-atomically implement the corresponding updates to the underlying files on ordinary file systems, SQLite implements a complex multi-journal mechanism [27]. Such application-level logging can interact pathologically with analogous mechanisms in the underlying file and storage systems [33]. Support for multi-file atomic updates in the file system can simplify and streamline applications that require this capability.

AdvFS supports multi-file atomic durability via its new "`syncv`" mechanism, which is implemented as an `ioctl` for compatibility with the stock Linux kernel. Our `syncv` achieves failure atomicity by leveraging the AdvFS journaling mechanism. AdvFS is a journaling file system that employs write-ahead logging to ensure the integrity of the file system. Modifications to the metadata are completely written to the journal before the actual changes are written to storage. The journal is written to storage at regular intervals. During crash recovery, AdvFS reads the journal to confirm file system transactions. All completed transactions are committed to storage and uncompleted transactions are undone. The number of uncommitted records in the journal, not the amount of data in the file system, determines the speed of recovery.

Our `syncv` takes as arguments an array of file descriptors opened with `O_ATOMIC` and the size of the array. Our `O_ATOMIC` implementation is the building block for implementing `syncv`. As noted previously `O_ATOMIC` deletes the existing clone and creates a new one at the time of `fsync`/`msync`. In order to make `syncv` atomic the delete operation on all of the files' clones must be atomic. This is achieved using AdvFS's journaling sub-system. Metadata modifications required to delete the clones are logged to the journal. The journaling sub-system ensures that all of these changes are atomically and durably committed. Apart from this the recovery for `syncv` is no different from single files opened with `O_ATOMIC`. Creation of new clones for the files need not be made atomic in `syncv` because the files and their new clones are mapped to the same storage. Unlike prior work on single-file atomic durability which uses journaling for capturing data changes [20], our multi-file atomic durability mechanism `syncv` uses the journal not for *application data* but only for the *metadata* changes needed to delete the clones. This metadata change is quite small and hence allows our approach to support a very large number of multiple file updates simultaneously. With the AdvFS default journal size of 128 MB, a single `syncv` call can atomically update at least 256 files under worst-case conditions. Configuring a larger journal will proportionately increase the number of files that `syncv` can accommodate.

The 256-file limitation stems from a combination of our current implementation of "clone delete," the worst case size of a single "clone delete," and the size of the AdvFS journal. AdvFS checkpoints the journal at every quadrant, which limits journal transaction size to 25% of the journal size. In a badly fragmented file system, the delete of a single file (or clone) could occupy 128 KB in the journal. So for a 128 MB journal, in the worst case our current implementation of `syncv` can atomically update $(128 \times 25\% \times 1024 \times 1024)/(128 \times 1024) = 256$ files. In principle we could support far more files per `syncv` by using Delayed Delete Lists (DDL). A DDL is a list maintained on non-volatile storage that is used to asynchronously delete files. If "clone delete" were to use DDL, then the journal footprint of each would be roughly 100 bytes and `syncv` would be able to handle hundreds of thousands of files.

It is important to understand that clones are taken of individual files only, *not* the entire file system nor any subtree thereof. Cloning an individual file involves creating a copy of the file's inode and an associated (hidden) dentry. These steps are atomic for the same reason that ordinary file creation is atomic: they are journaled. Atomic update of an individual file involves first flushing changes to the file, then unlinking its clone, then creating a new clone; these operations are journaled separately and sequentially, so partial or full recovery of these three sequential but disjoint operations always leaves the system in a consistent state. Our `syncv` mechanism, which operates on multiple files, obtains atomicity by leveraging the file system journal to ensure, in REDO-log fashion, that all of the per-file atomic updates in a specified bundle are (eventually) performed.

4 Evaluation

We verify that AdvFS `O_ATOMIC` does indeed protect the integrity of application data across updates in the presence of crashes (Section 4.1), and we compare the performance of our solution to the CMADD problem with existing alternatives (Section 4.2).

4.1 Correctness

The `O_ATOMIC` data integrity guarantees rely upon two realistic assumptions about underlying storage systems. First, it must be possible to commit data to durable media synchronously, which means that volatile write caches in storage hardware must include enough standby power to rescue their contents to durable media if power fails. Second, we assume that writes of 512-byte sectors are atomic. Given these preconditions, the `O_ATOMIC` feature of AdvFS should protect application data integrity as advertised. We verify that it does so by injecting two types of failure: crash points and power interruptions.

Crash points are manually inserted into the AdvFS source code where the developers believe crashes are most likely to cause trouble, e.g., before, during, and after atomic operations. When a particular crash point is externally activated in a running instance of AdvFS, the result is an immediate storage system shutdown followed by a kernel panic, triggered from the specified crash point in the file system source code. We complement crash-point testing with sudden whole-system power interruptions, because the former test specific developer hypotheses about recovery whereas the latter strive to uncover surprising scenarios. Our power outage tests employ a scriptable device that suddenly cuts power to a computer without warning—the same effect as physically unplugging the machine's power cord.

The goal of both crash-point testing and power interruptions is to cause "impossible" post-crash corruption. Our test suites repeatedly and intensively modify files using `write` or `mmap`/STORE then call `fsync` or `msync`, respectively, or `syncv`. The patterns of data modifications are such that a defective implementation of `O_ATOMIC` would likely introduce obvious evidence of corruption following a crash. Finally, we trigger crashes while our test suites are running. Our crash point-tests ran against an enterprise-class RAID controller and our whole-system power interruptions ran on an enterprise-class SSD, both of which are described in Section 4.2. AdvFS successfully survived over 400 power interruptions and dozens of crash-point tests, with no evidence of application data corruption. The expected kind of application data corruption *does* occur when we inject failures if `O_ATOMIC` is *not* used. For example, crashes during append leave partial data appended and crashes during `seek`/`write` sequences leave partial updates.

4.2 Performance

On file system benchmarks such as IOZONE [6], postmark [18] and MDTest [16], AdvFS performance is competitive with other well-known file systems such as ext3, ext4, and XFS; by most performance measures AdvFS is within $\pm 10\%$ of other file systems. Due to space limitations we omit these comparisons and focus on the performance of atomic file updates.

We evaluate the performance of the new `O_ATOMIC` feature via microbenchmarks that mimic a common use case of `fsync` and also via "mesobenchmarks" that compare a simple transactional key-value store built atop failure-atomic `msync` with well-known alternatives. Prior literature has documented the ease with which failure-atomic `writev`/`msync` can be retrofitted onto complex, mature, production software to improve resilience and performance [3, 20, 29].

We ran our tests on two systems: a workstation and an enterprise server. The workstation has two quad-core 2.4 GHz Xeon E5620 processors and 12 GB of 1333 MHz DRAM and ran Linux kernel 2.6.32. We installed AdvFS on the workstation's enterprise-grade 120 GB SATA drive on a 3 Gbps controller. The SSD is powerfail-safe because its write cache is backed by a supercapacitor. Prior to our experiments we "burned in" the SSD by writing to the device at least 180 GB of data (i.e.,

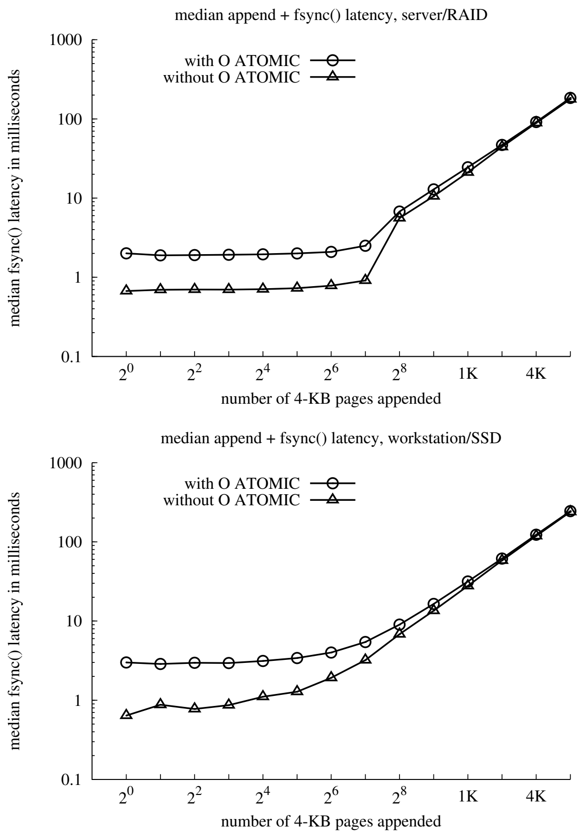


Figure 2: Microbenchmark results.

$1.5\times$ its rated capacity). Our enterprise server had twelve 1.8 GHz Xeon E5-2450L cores and 92 GB of DRAM; its storage controller had a 1 GB battery-backed cache configured as 90% write cache and a 1 TB 7200 RPM SAS hard drive.

What overhead does `O_ATOMIC` entail compared to operations on files opened without this flag? Our microbenchmark addresses this question in the context of a common use case: using `write` to append data to a file followed by `fsync` to commit the amendments to durable media. Figure 2 presents median `fsync` latencies for this operation on files opened with and without `O_ATOMIC` on our two test machines. Our results show that `O_ATOMIC` carries a constant overhead on the order of 2 ms, which is clearly visible in Figure 2 for appends of up to 2^7 pages (512 KB). This overhead occurs because the current implementation of `O_ATOMIC` in AdvFS performs an uncached storage read in connection with creating an inode for the clone when `fsync` is called. This is a simplification in our current implementation and clone creation times could be reduced by copying in-core state rather than reading from storage.

	Server/RAID			Workstation/SSD		
	insert	replace	delete	insert	replace	delete
STL `<map>`/AdvFS	1.996	2.488	2.919	1.655	2.022	2.395
Kyoto Cabinet 1.2.76	4.711	2.990	4.660	4.088	2.590	4.007
SQLite 3.7.14.1	2.394	2.524	2.433	2.374	2.611	2.435
LevelDB 1.6.0	0.629	0.626	0.615	0.641	0.640	0.633

Table 1: Mesobenchmarks: Mean per-operation timings (milliseconds).

For large modifications, the roughly constant overhead of `O_ATOMIC` becomes negligible (approximately 2–3.5%) and we approach the rated write bandwidth of the SSD. In other words, the price we pay for failure-atomicity is modest for large updates.

Our "mesobenchmark" repeats the experiment in Section 5.3 of our original failure-atomic `msync` paper [20], which compares four transactional key-value stores: SQLite [26], LevelDB [14], Kyoto Cabinet [13], and a fourth contender implemented as a C++ Standard Template Library (STL) `<map>` container that stores data in a persistent heap backed by a file opened with `O_ATOMIC` and updated with `msync`. Each of these key-value stores performed the following transactional operations on one thousand keys: first, insert all keys paired with random 1 KB values; next, replace the value associated with each key with a different random value; and finally, delete all of the keys, for a total of three thousand transactions. Each of the above three steps visits keys in a different random order, i.e., we randomly permute the universe of keys before each step.

Table 1 presents our results, which are comparable to those in our earlier work. LevelDB wins hands down, with the STL `<map>` atop a memory-mapped file updated with AdvFS failure-atomic `msync` placing second. This is easy to understand: The red-black tree beneath an STL `<map>` makes no attempt to minimize the number of memory pages it modifies, which strongly influences the performance of STL/AdvFS; by contrast, LevelDB implements failure-atomic updates with carefully crafted, compact log file writes. Our simple `<map>`-based key-value store shows that a persistent heap based on atomic file update can very easily slide beneath a rich, full-featured library of in-memory data structures and algorithms—which takes roughly a dozen lines of code in the present case. The net result is to transform software with no failure resilience whatsoever into software that can withstand process crashes, OS kernel panics, and power outages. Our AdvFS-fortified `<map>` furthermore achieves better performance than two far more complex platforms designed to provide failure resilience with good performance. Our experience convinces us that failure-atomic file update enables dramatically simplified application software whose performance rivals all but expertly streamlined code.

5 Conclusions

We have shown that a mechanism for consistent modification of application durable data (CMADD) can be implemented straightforwardly atop per-file cloning, a feature already available in AdvFS and in several other modern FSes. Our implementation of `O_ATOMIC` exposes a simple interface to applications, makes file modifications via both `write` and `mmap` failure-atomic, avoids double writes, and supports very large transactional updates of application data. Furthermore, our `syncv` implementation supports failure-atomic updates of application data in *multiple* files. Our empirical results show that the `O_ATOMIC` implementation in AdvFS preserves the integrity of application data across updates in the presence of both surgically inserted crashes and sudden power interruptions. Our performance evaluation shows that `O_ATOMIC` carries tolerable overheads, particularly for large atomic updates.

Implementing a CMADD mechanism in a file system facilitates adoption because it requires neither special hardware nor modified OS kernels. We believe that file systems should implement simple, general, robust CMADD mechanisms, that many applications would exploit such a feature if it were widely available, and that `O_ATOMIC` and `syncv` are convenient interfaces.

Acknowledgments

We thank Joe Tucek and Haris Volos for helpful advice and feedback on early drafts of this paper. The anonymous FAST reviewers and our shepherd, Raju Rangaswami, provided valuable suggestions and guidance for the final version. We thank HP Storage manager Subbi Mudigere for the support and resources that enabled us to implement the research concept of failure-atomic file update in AdvFS.

References

[1] Tru64 AdvFS technology. Retrieved 17 September 2014 from `http://advfs.sourceforge.net/`.

[2] Berkeley Database (BDB). `http://www.oracle.com/us/products/database/berkeley-db/overview/index.html`.

[3] A. Blattner, R. Dagan, and T. Kelly. Generic crash-resilient storage for Indigo and beyond. Technical Report HPL-2013-75, Hewlett-Packard Laboratories, Nov. 2013. `http://www.hpl.hp.com/techreports/2013/HPL-2013-75.pdf`.

[4] Btrfs file system, Sept. 2014. `https://btrfs.wiki.kernel.org/index.php/Main_Page`.

[5] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Optimistic crash consistency. In *SOSP*, 2013.

[6] D. Capps. IOzone filesystem benchmark. `http://www.iozone.org/`.

[7] Apple `exchangedata(2)` manual page. Retrieved 22 September 2014 from `https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man2/exchangedata.2.html`.

[8] FusionIO. NVM primitives library, Feb. 2014. See in particular the `nvm_batch_atomic_operations` at `http://opennvm.github.io/nvm-primitives-documents/`.

[9] J. Guerra, L. Mármol, D. Campello, C. Crespo, R. Rangaswami, and J. Wei. Software persistent memory. In *USENIX Annual Technical Conference*, 2012. `https://www.usenix.org/system/files/conference/atc12/atc12-final70.pdf`.

[10] G. Haff. The many lives of AdvFS, June 2008. Retrieved 17 September 2014 from `http://www.cnet.com/news/the-many-lives-of-advfs/`.

[11] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A file is not a file: Understanding the I/O behavior of Apple desktop applications. In *SOSP*, 2011. `http://doi.acm.org/10.1145/2043556.2043564`.

[12] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *USENIX Winter Technical Conference*, 1994. `http://dl.acm.org/citation.cfm?id=1267074.1267093`.

[13] Kyoto Cabinet: a straightforward implementation of DBM. `http://fallabs.com/kyotocabinet/`.

[14] Google's LevelDB key-value store. `https://github.com/google/leveldb`.

[15] L. Marmol, S. Sundararaman, N. Talagala, R. Rangaswami, S. Devendrappa, B. Ramsundar, and S. Ganesan. NVMKV: A scalable and lightweight flash aware key-value store. In *HotStorage*, June 2014. `https://www.usenix.org/system/files/conference/hotstorage14/hotstorage14-paper-marmol.pdf`.

[16] Mdtest: A synthetic benchmark for file systems metadata operations. `sourceforge.net/projects/mdtest/`.

[17] Microsoft Developer Network. Alternatives to using transactional NTFS. Retrieved 17 September 2014 from `http://msdn.microsoft.com/en-us/library/hh802690.aspx`.

[18] Network Appliance. Postmark: A New Filesystem Benchmark, Technical Report TR3022, Network Appliance, 1997. `www.netapp.com/tech_library/3022.html/`.

[19] T. Neward. Object-relational mapping: The Vietnam of computer science, June 2006. `http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx`.

[20] S. Park, T. Kelly, and K. Shen. Failure-atomic `msync()`: A simple and efficient mechanism for preserving the integrity of durable data. In *EuroSys*, 2013. `http://doi.acm.org/10.1145/2465351.2465374`.

[21] Z. Peterson and R. Burns. Ext3Cow: A time-shifting file system for regulatory

compliance. *ACM Transactions on Storage*, 1(2), May 2005. http://doi.acm.org/10.1145/1063786.1063789.

[22] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating system transactions. In *SOSP*, 2009. http://www.cs.utexas.edu/users/witchel/pubs/porter09sosp-txos.pdf.

[23] Microsoft Windows ReplaceFile function. Retrieved 22 September 2014 from http://msdn.microsoft.com/en-us/library/aa365512.aspx.

[24] M. Saxena, M. A. Shah, S. Harizopoulos, M. M. Swift, and A. Merchant. Hathi: Durable transactions for memory using flash. In *Non-Volatile Memories Workshop*, Mar. 2012. http://pages.cs.wisc.edu/~swift/papers/nvmw12_hathi.pdf.

[25] R. P. Spillane, S. Gaikwad, M. Chinni, E. Zadok, and C. P. Wright. Enabling transactional file access via lightweight kernel extensions. In *FAST*, 2009. https://www.usenix.org/legacy/event/fast09/tech/full_papers/spillane/spillane.pdf.

[26] SQLite database library. http://www.sqlite.org/.

[27] SQLite multi-file atomic commit (Section 5). http://www.sqlite.org/atomiccommit.html.

[28] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *USENIX Annual Technical Conference*, 1996. http://dl.acm.org/citation.cfm?id=1268299.1268300.

[29] N. Talagala. Atomic writes accelerate MySQL performance, Oct. 2011. http://www.fusionio.com/blog/atomic-writes-accelerate-mysql-performance/.

[30] The Open Group. *Portable Operating System Interface (POSIX) Base Specifications, Issue 7, IEEE Standard 1003.1*. IEEE, 2008.

[31] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *ASPLOS*, 2011. http://doi.acm.org/10.1145/1950365.1950379.

[32] Symantec Veritas VxFS file system. Retrieved 23 September 2014 from https://sort.symantec.com/public/documents/sfha/6.0/aix/manualpages/html/man/storage_foundation_for_databases_tools/html/man1m/vxsfadm-filesnap.1m.html.

[33] J. Yang, N. Plasson, G. Gillis, N. Talagala, and S. Sundararaman. Dont stack your log on my log. In *USENIX Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW)*, 2014. https://www.usenix.org/system/files/conference/inflow14/inflow14-yang.pdf.

[34] S. Yoo, C. Killian, T. Kelly, H. K. Cho, and S. Plite. Composable reliability for asynchronous systems. In *USENIX Annual Technical Conference*, 2012. https://www.usenix.org/system/files/conference/atc12/atc12-final206-7-20-12.pdf.

[35] OpenZFS, Sept. 2014. http://www.open-zfs.org/wiki/Main_Page.

[36] 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 *OSDI*, Oct. 2014. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/zheng_mai.

A Tale of Two Erasure Codes in HDFS

Mingyuan Xia[*], Mohit Saxena[†], Mario Blaum[†] and David A. Pease[†]
[*]*McGill University*, [†]*IBM Research Almaden*
mingyuan.xia@mail.mcgill.ca,{msaxena,mmblaum,pease}@us.ibm.com

Abstract

Distributed storage systems are increasingly transitioning to the use of erasure codes since they offer higher reliability at significantly lower storage costs than data replication. However, these codes tradeoff recovery performance as they require multiple disk reads and network transfers for reconstructing an unavailable data block. As a result, most existing systems use an erasure code either optimized for storage overhead or recovery performance.

In this paper, we present HACFS, a new erasure-coded storage system that instead uses *two different erasure codes* and dynamically adapts to workload changes. It uses a fast code to optimize for recovery performance and a compact code to reduce the storage overhead. A novel conversion mechanism is used to efficiently upcode and downcode data blocks between fast and compact codes. We show that HACFS design techniques are generic and successfully apply it to two different code families: Product and LRC codes.

We have implemented HACFS as an extension to the Hadoop Distributed File System (HDFS) and experimentally evaluate it with five different workloads from production clusters. The HACFS system always maintains a low storage overhead and significantly improves the recovery performance as compared to three popular single-code storage systems. It reduces the degraded read latency by up to 46%, and the reconstruction time and disk/network traffic by up to 45%.

1 Introduction

Distributed storage systems storing multiple petabytes of data are becoming common today [4, 25, 2, 15]. These systems have to tolerate different failures arising from unreliable components, software glitches, machine reboots, and maintenance operations. To guarantee high reliablity and availablity despite these failures, data is replicated across multiple machines and racks. For example, the Google File System [11] and the Hadoop Distributed File System [4] maintain three copies of each data block. Although disk storage seems inexpensive today, replication of the entire data footprint is simply infeasible at massive scales of operation. As a result, most large-scale distributed storage systems are transitioning to the use of erasure codes [3, 2, 15, 20], which provide higher reliability at significantly lower storage costs.

The trade-off for using erasure codes instead of replicating data is performance. If a data block is three-way replicated, it can be reconstructed by copying it from one of its available replicas. However, for an erasure-coded system, reconstructing an unavailable block requires fetching multiple data and parity blocks within the code stripe, which results in significant increase in disk and network traffic. Recent measurements on a Facebook's data warehouse cluster [19, 20] storing multiple petabytes of erasure-coded data, required a median of more than 180 Terabytes of data transferred to recover from 50 machine-unavailability events per day.

This increase in the amount of data to be read and transferred during recovery for an erasure-coded system results in two major problems: *high degraded read latency* and *longer reconstruction time*. First, a read to an unavailable block requires multiple disk reads, network transfers and compute cycles to decode the block. The application accessing the block waits for the entire duration of this recovery process, which results in higher latencies and degraded read performance. Second, a failed or decommissioned machine, or a failed disk results in significantly longer reconstruction time than in a replicated system. Although, the recovery of data lost from a failed disk or machine can be performed in the background, it severely impacts the total throughput of the system as well as the latency of degraded reads during the reconstruction phase.

As a result, the problem of reducing the overhead of recovery in erasure-coded systems has received signifi-

[*]Work done as an intern at IBM Research Almaden

cant attention in the recent past both in theory and practice [19, 2, 20, 15, 24, 3, 14, 26]. Most of the solutions tradeoff between two dimensions: storage overhead and recovery cost. Storage overhead accounts for the additional parity blocks for a coding scheme. Recovery cost is the total number of blocks required to reconstruct a data block after failure.

In general, most production systems use a *single erasure code*, which either optimizes for recovery cost or storage overhead. For example, Reed-Solomon [21] is a popular family of codes used in Google's ColossusFS [2], Facebook's HDFS [3], and several other storage systems [20, 25, 16]. The Reed-Solomon code used in ColossusFS has a storage overhead of 1.5x, while it requires six disk reads and network transfers to recover a lost data block. In contrast, the Reed-Solomon code used in HDFS reduces the storage overhead to 1.4x, but has a recovery cost of ten blocks. The other popular code family is the Local Reconstruction Codes (LRC) [15, 24, 26], and has similar tradeoffs.

In this paper, we present Hadoop Adaptively-Coded Distributed File System (*HACFS*), a new erasure-coded storage system, which instead uses *two different erasure codes* from the same code family. It uses a *fast code* with low recovery cost and a *compact code* with low storage overhead. It exploits the data access skew observed in Hadoop workloads [9, 7, 22, 5] to decide the initial encoding of data blocks. The HACFS system uses the fast code to encode a small fraction of the frequently accessed data and provide overall low recovery cost for the system. It uses the compact code to encode the majority of less frequently accessed data blocks and maintain a low and bounded storage overhead.

After initial encoding, the HACFS system dynamically adapts to workload changes by using two novel operations to convert data blocks between the fast and compact codes. *Upcoding* blocks initally encoded with fast code into compact code enables the HACFS system to reduce the storage overhead. Similarly, *downcoding* data blocks from compact code to fast code representation lowers the overall recovery cost of the HACFS system. The upcode and downcode operations are very efficient and only update the associated parity blocks while converting blocks between the two codes.

We have designed and implemented HACFS as an extension to the Hadoop Distributed File System [3]. We find that adaptive coding techniques in HACFS are generic and can be applied to different code families. We successfully implement adaptive coding in HACFS with upcode and downcode operations designed for two different code families: Product codes [23] and LRC codes [15, 24, 26]. In both cases, HACFS with adaptive coding using two codes outperforms HDFS with a single Reed-Solomon [2, 3] or LRC code [15]. We evaluate our design on an HDFS cluster with workload distributions obtained from production environments at Facebook and four different Cloudera customers [9].

The main contributions of this paper are as follows:

- We design HACFS, a new erasure-coded storage system that adapts to workload changes by using two different erasure codes - a fast code to optimize recovery cost of degraded reads and reconstruction of failed disks/nodes, and a compact code to provide low and bounded storage overhead.
- We design a novel conversion mechanism in HACFS to efficiently up/down-code data blocks between the two codes. The conversion mechanism is generic and we implement it for two code families – Product and LRC codes – popularly used in distributed storage systems.
- We implement HACFS as an extension to HDFS and demonstrate its efficacy using two case studies with Product and LRC family of codes. We evaluate HACFS by deploying it on a cluster with real-world workloads and compare it against three popular single code systems used in production. The HACFS system always maintains a low storage overhead, while improving the degraded read latency by 25-46%, reconstruction time by 14-44%, and network and disk traffic by 19-45% during reconstruction.

The remainder of the paper is structured as follows. Section 2 motivates HACFS by describing the different tradeoffs for erasure-coded storage systems and HDFS workloads. Section 3 and 4 present the detailed description of HACFS design and implementation. Finally, we evaluate HACFS design techniques in Section 5, and finish with related work and conclusions.

2 Motivation

In this section, we describe the different failure modes and recovery methods in erasure-coded HDFS [3]. We discuss how the use of erasure codes within HDFS reduces storage overhead, however it increases the recovery cost. This motivates the need to design HACFS, which exploits the data access characteristics of Hadoop workloads to achieve better recovery cost and storage efficiency than the existing HDFS architecture.

Failure Modes and Recovery in HDFS. HDFS has different failure modes, for example, block failure, disk failure, and a decommisioned or failed node. The causes of these failures may be diverse such as hardware failures, software glitches, maintenance operations, rolling upgrades that take certain percentage of nodes offline, and hot-spot effects that overload particular disks. Most of these failures typically result in the unavailability of a single block within an erasure code stripe. An erasure

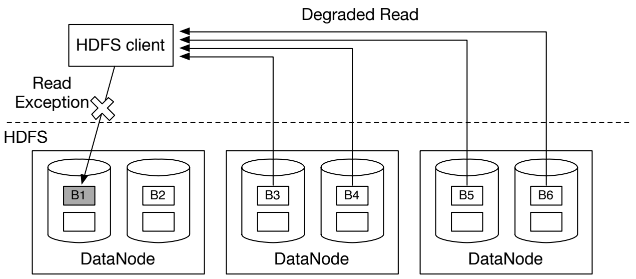


Figure 1: **Degraded reads for HDFS block failure:** *The figure shows a degraded read for an HDFS client reading an unavailable block B_1. The HDFS client retrieves available data and parity blocks and decodes the block B_1.*

code stripe is composed of multiple data blocks striped across different disks or nodes in an HDFS cluster. Over 98% of all failure modes in Facebook's data-warehouse and other production HDFS clusters require recovery of a single block failure [20, 19]. Another 1.87% have two blocks missing, and just less than 0.05% are three or more block failures. As a result, most recent research on erasure-coded storage systems has focused on reducing the recovery cost of single block failures [24, 20, 15, 16].

The performance of an HDFS client or a MapReduce task can be affected by HDFS failures in two ways: degraded reads and reconstruction of an unavailable disk or node. Figure 1 shows a degraded read for an HDFS client reading an unavailable data block B_1, which returns an exception. The HDFS client recovers this block by first retrieving the available data and parity blocks within the erasure-code stripe from other DataNodes. Next, the HDFS client decodes the block B_1 from the available blocks. Overall, the read to a single block B_1 is delayed or degraded by the time it takes to perform several disk reads and network transfers for available blocks, and the time for decoding. Reed-Solomon codes used in two production filesystems - Facebook's HDFS [3] and Google ColossusFS [2] - require between 6-10 network transfers and several seconds for completing one degraded read (see Section 5.3).

A failed disk/node or a decomissioned node typically requires recovery of several lost data blocks. When a DataNode or disk failure is detected by HDFS, several MapReduce jobs are launched to execute parallel recovery of lost data blocks on other live DataNodes. HDFS places data blocks in an erasure-code stripe on different disks and nodes. As a result, the reconstruction of most disk and node failures effectively requires recovery of several single block failures similar to degraded reads.

Figure 2 shows the network transfers required by the reconstruction job running on live DataNodes. An HDFS client trying to access lost blocks B_1 and B_2 during the reconstruction phase encounters degraded reads. Overall, the reconstruction of lost data from a failed disk or node results in several disk reads and network transfers, and can take from tens of minutes to hours for complete recovery (see Section 5.4).

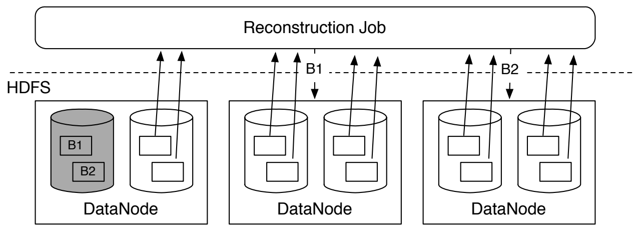


Figure 2: **Reconstruction for HDFS node/disk failure or decomissioned nodes:** *The figure shows a reconstruction MapReduce job launched to recover from a disk failure on an HDFS DataNode. The reconstruction job executes parallel recovery of the lost blocks B_1 and B_2 on other live DataNodes by retrieving available data and parity blocks. An HDFS client accessing a lost block encounters degraded reads during the reconstruction phase.*

Erasure Coding Tradeoffs. Figure 3 show the recovery cost and storage overhead for Reed-Solomon family of codes widely used in production systems [2, 3, 25]. In addition, it also shows the recovery cost and storage overhead of three popular erasure-coded storage systems: Google ColossusFS [2], Facebook HDFS [3], and Microsoft Azure Storage [15].

Google ColossusFS and Facebook HDFS use two different Reed-Solomon codes - $RS(6,3)$ and $RS(10,4)$- that encode six and ten data blocks within an erasure-code stripe with three and four parity blocks respectively. As a result, they have a recovery cost of six and ten blocks, and storage overheads of 1.5x and 1.4x respectively. Microsoft Azure Storage uses an LRC code - LRC_{comp}, which reduces the storage overhead to 1.33x and has a similar recovery cost of six blocks as Google ColossusFS. It encodes twelve data blocks with two global and two local parity blocks (see Section 3.3 for more detailed description on LRC codes). In contrast, three-way data replication provides recovery cost of one block, but a higher storage overhead of 3x. In general, most erasure-codes including Reed-Solomon and LRC codes trade-off between recovery cost and storage overhead, as shown in Figure 3.

In this work, we focus on the blue region in Figure 3 to achieve recovery cost for HACFS less than that of both Reed-Solomon and LRC codes used in ColossusFS and Azure. We further exploit the data access skew in Hadoop workloads to maintain a low storage overhead for HACFS and keep it bounded between the storage overheads of these two systems.

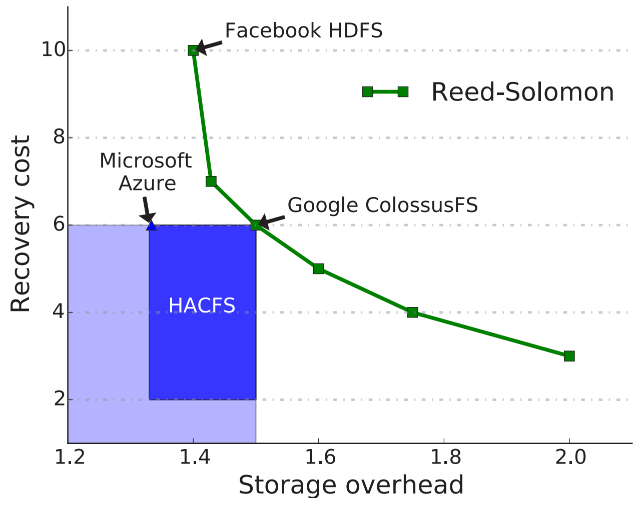


Figure 3: **Recovery Cost vs. Storage Overhead:** *The figure shows the tradeoff between recovery cost and storage overhead for the popular Reed-Solomon family of codes. It also shows three production storage systems each using single erasure code.*

Data Access Skew. Data access skew is a common characteristic of Hadoop storage workloads [9, 7, 22, 5]. Figure 4 shows the frequency of data accessed in production Hadoop clusters at Facebook and four different Cloudera customers [9]. All workloads show skew in data access frequencies. The majority of the *data volume* is cold and accessed only a few times. Similarly, the majority of the *data accesses* go to a small fraction of data, which is hot. In addition, HDFS does not allow in-place block updates or overwrites. As a result, the read accesses primarily characterize this data access skew.

Why HACFS? The HACFS design aims to achieve the following goals:

- *Fast degraded reads* to reduce the latency of reads when accessing lost or unavailable blocks.
- *Low reconstruction time* to reduce the time for recovering from failed disks/nodes or decommissioned nodes, and the associated disk and network traffic.
- *Low storage overhead* that is bounded under practical system constraints and adjusted based on workload requirements.

As shown in Figure 3, the use of a single erasure code tradesoff recovery cost for storage overhead. To achieve the above design goals, the HACFS system uses a combination of two erasure codes and exploits the data access skew within the workload. We next describe HACFS design in more detail.

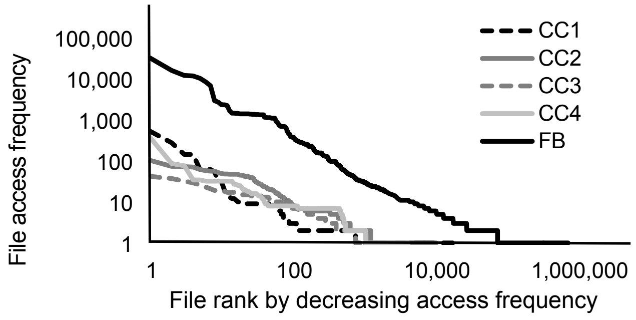


Figure 4: **Data Access Skew in Hadoop Workloads:** *The figure shows the data access distributions of Hadoop workloads collected from production clusters at Facebook (FB) and four different Cloudera customers (CC-1,2,3,4). Both axes are on log scale.*

3 System Design

In this section, we first describe how the HACFS system adapts between the two different codes based on workload characteristics to reduce recovery cost and storage overhead. We next discuss the application of HACFS's adaptive coding to two different code families: Product codes with low recovery cost and LRC codes with low storage overhead.

3.1 Adaptive Coding in HACFS

The HACFS system is implemented as an extension to the HDFS-RAID module [3] within HDFS. We show our extensions to HDFS-RAID as three shaded components in Figure 5. The adaptive coding module maintains the system states of erasure-coded data and manages state transitions for ingested and stored data. It also interfaces with the erasure coding module, which implements the different coding schemes.

System States. The adaptive coding module of HACFS manages the system state. The system state tracks the following *file state* associated with each erasure-coded data file: file size, last modification time, read count and coding state. The file size and last modification time are attributes maintained by HDFS, and used by HACFS to compute the total data storage and write age of the file. The adaptive coding module also tracks the read count of a file, which is the total number of read accesses to the file by HDFS clients. The coding state of a file represents if it is three-way replicated or the erasure coding scheme used for it. The file state can be updated on a create, read or write operation issued to the file from an HDFS client.

The adaptive coding module also maintains a *global state*, which is the total storage used for data and parity. Every block in a replicated data file is replicated at three different nodes in the HDFS cluster and the two replicas account for the parity storage. In contrast, every

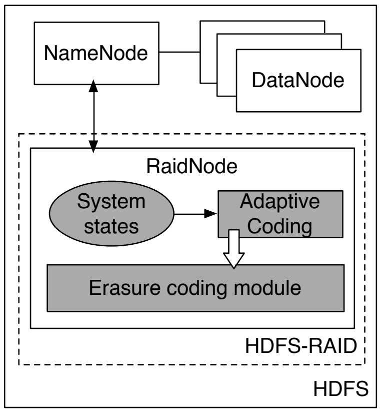


Figure 5: **HACFS Architecture:** *The figure shows the different components of the HACFS architecture implemented as extensions to HDFS in the shaded modules.*

block in an erasure-coded data file has exactly one copy. Each erasure-coded file is split into different erasure code stripes, with blocks in each stripe distributed across different nodes in the HDFS cluster. Each erasure-coded data file also has an associated parity file whose size is determined by the coding scheme. The global state of the system is updated periodically when the adaptive coding module initiates state transitions for erasure-coded files. A state transition corresponds to a change in the coding state of a file and is invoked by using the following interfaces to the erasure-coding module.

Coding Interfaces. As shown in Table 1, the *erasure coding module* in HACFS system exports four major interfaces for coding data: encode, decode, upcode and downcode. The *encode* operation requires a data file and coding scheme as input, and generates a parity file for all blocks in the data file. The *decode* operation is invoked on a degraded read for a block failure or as part of the reconstruction job for a disk or node failure. It also requires the index of the missing or corrupted block in a file, and reconstructs the lost block from the remaining data and parity blocks in the stripe using the input coding scheme.

The adaptive coding module invokes upcode and downcode operations to adapt with workload changes and convert a data file representation between the two coding schemes. As we show later in Section 3.2 and 3.3, both of these conversion operations only update the associated parity file when changing the coding scheme of a data file. The *upcode* operation transforms data from a fast code to a compact code representation, thus reducing the size of the parity file to achieve lower storage overhead. It does not require reading the data file and is a parity-only transformation. The *downcode* operation transforms from a compact code to a fast code representation, thus reducing the recovery cost. It requires reading both data and parity files, but only changes the parity file. We next explain how HACFS uses these interfaces for transitioning files between different coding schemes based on the file state and global system state.

Function	**Input**	**Output**
encode	data file, codec	parity file
decode	data file, parity file, codec, lost block index	recovered block
upcode	parity file, original fast codec, new compact codec	parity file encoded with compact codec
downcode	data file, parity file, original compact codec, new fast codec	parity file encoded with fast codec

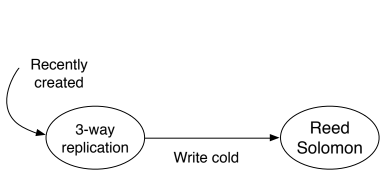
Table 1: **The HACFS Erasure Coding Interfaces**

State Transitions. The HACFS system extends HDFS-RAID to use different erasure coding schemes for files with different read frequency, thus achieving the low recovery cost of a fast code and the low storage overhead of a compact code. We first describe the basic state machine used in HDFS-RAID and then elaborate on the HACFS extensions.

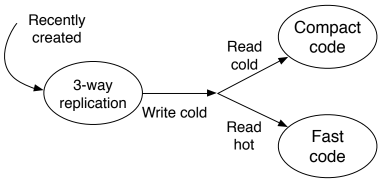
As shown in Figure 6(a), a recently created file in HDFS-RAID is classified as write hot based on its last modified time and therefore three-way replicated. The HDFS-RAID process (shown as RaidNode in Figure 5) scans the file system periodically to select write cold files for erasure coding. It then schedules several MapReduce jobs to encode all such candidate files with a Reed-Solomon code [3]. After encoding, the replication level of these files is reduced to one and the coding state changes to Reed-Solomon. As HDFS only supports appends to files, a block is never overwritten and these files are only read after being erasure-coded.

Figure 6(b) shows the first extension of the HACFS system. It replicates write hot files similar to HDFS-RAID. In addition, HACFS also accounts for the read accesses to data blocks in a file. All write cold files are further classified based on their read counts and encoded with either of the *two different erasure codes*. Read hot files with a high read count are encoded with a *fast code*, which has a low recovery cost. Read cold files with a low read count are encoded with a *compact code*, which has a low storage overhead.

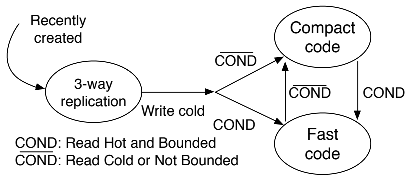
However, a read cold file can later get accessed and turn into a read hot file, thereby requiring low recovery cost. Similarly, encoding all files with the fast code may result in a higher total storage overhead for the system. As a result, the HACFS system needs to adapt to the workload by converting files between fast and compact codes (as shown in Figure 6(c)). The conversion for a file is guided by its own file state (read count) as well as the global system state (total storage). When the total storage consumed by data and parity blocks exceeds a configured system *storage bound*, the HACFS system se-



(a) Single Code in HDFS



(b) Two Codes in HACFS



(c) Adaptive Coding in HACFS

Figure 6: **Execution States:** *The figure shows the functional state machines for HDFS and two extensions for HACFS. Transitions between different states are triggered by the adaptive coding module, which invokes the coding interface exported by the erasure-coding module in HACFS.*

	PC_{fast}	PC_{comp}	LRC_{fast}	LRC_{comp}
DRC	2	5	2	6
RC	2	5	3.25	6.75
SO	1.8x	1.4x	1.66x	1.33x
MTTF	1.4×10^{12}	2.1×10^{11}	6.1×10^{11}	8.9×10^{10}

Table 2: **Fast and Compact Codes**: This table shows the two codes in the Product and LRC code families used for adaptive coding in HACFS (*DRC*: Degraded Read Cost, *RC*: Reconstruction Cost, *SO*: Storage Overhead, *MTTF*: Mean-Time-To-Failure in years).

lects some files encoded with fast code and upcodes them to the compact code. Similarly, it selects some replicated files and encodes them directly into the compact code. The HACFS system selects these files by first sorting them based on their read counts and then upcodes/encodes the files with lowest read counts into compact code to make the total storage overhead bounded again.

The downcode operation transitions a file from compact to fast code. As a result, it reduces the recovery cost of a future degraded read to a file, which was earlier compact-coded but has been recently accessed. As shown in Figure 4, the data access skew for Hadoop workloads result in a small fraction of read hot files and large fraction of read cold files. This skew allows HACFS to reduce the recovery cost by encoding/downcoding the read hot files with a fast code and reduces the storage overhead by encoding/upcoding a large fraction of read cold files with a compact code.

Fast and Compact Codes. The adaptive coding techniques in HACFS are generic and can be applied to different code families. We demonstrate its application to two code families: Product codes [23] with low recovery cost and LRC codes [15, 24] with low storage overhead. Table 2 shows the three major characteristics useful for selecting fast and compact codes from a code family. The fast code must have a low recovery cost for degraded reads and reconstruction. The compact code must have a low storage overhead. Finally, the reliability of both codes measured in terms of mean-time-to-failure for data loss must be greater than that for three-way replication (3.5×10^9 years) [13]. In addition, the HACFS system requires a storage bound, which can be set from the practical requirements of the system or can be optimally tuned close to the storage overhead of the compact code. We use a storage bound of 1.5x with Product codes and 1.4x with LRC codes in the two case studies of the HACFS system.

We next describe the design of the *erasure coding module* in HACFS for Product and LRC codes in Section 3.2 and 3.3 respectively.

3.2 Product Codes

We now describe the construction and coding interfaces of Product codes used in the HACFS system.

Encoding and Decoding. Figure 7 shows the construction of a Product code, PC_{fast} or $PC(2 \times 5)$, which has a stripe with two rows and five columns of data blocks. The encode operation for PC_{fast} retrieves the ten data blocks from different locations in the HDFS cluster and generates two horizontal, five vertical and one global parity. The horizontal parities are generated by transferring the five data blocks in each row and performing an XOR operation on them. A vertical parity only requires two data block transfers in a column. The global parity can be constructed as an XOR of the two horizontal parities. The decode operation for a Product code is invoked on a block failure. A single failure in any data or parity block of the PC_{fast} code requires only two block transfers from the same column to reconstruct it.

As a result, the PC_{fast} code can achieve a very low recovery cost of two block transfers at the cost of a high storage overhead of eight parity blocks for ten data blocks (1.8x). We choose the PC_{comp} or $PC(6 \times 5)$ as the compact code (see Figure 7), which provides a lower storage overhead of 1.4x and higher recovery cost of five block transfers (see Table 2). In addition, both fast and compact Product codes have reliability better than three-way replication. We select a storage bound of 1.5x for the HACFS system with these Product codes since it is close to the storage overhead of the PC_{comp} code. This bound also matches the practical limits prescribed by the Google ColossusFS [2], which uses the Reed-Solomon $RS(6, 3)$ code similarly optimized for recovery cost.

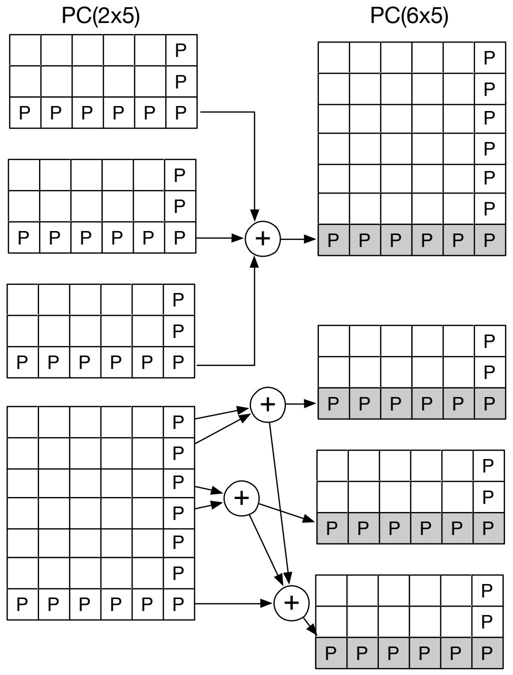


Figure 7: **Product Code - Upcode and Downcode Operations:** *The figure shows the upcode and downcode operations on the data and parity blocks for Product codes. The shaded horizontal parity blocks in the output code are only computed, and the remaining blocks remain unchanged from the input code.*

Upcoding and Downcoding. Figure 7 shows upcoding from PC_{fast} to PC_{comp} and downcoding from PC_{comp} to PC_{fast} codes. Upcode is a very efficient *parity-only conversion* operation for Product codes. All data and vertical parity blocks remain unchanged in upcoding from the PC_{fast} to PC_{comp} code. Further, it only performs XOR over the old horizontal and global parity blocks of the three PC_{fast} codes to compute the new horizontal and global parity blocks of the PC_{comp} code. As a result, the upcode operation does not require any network transfers of the data blocks from the three PC_{fast} codes to compute the new parities in the PC_{comp} code.

The downcode operation converts a PC_{comp} code into three PC_{fast} codes. Only the horizontal and global parities change between the PC_{comp} code and the three PC_{fast} codes. However, computing the horizontal and global parities in the first two PC_{fast} codes requires network transfers and XOR operations over the data blocks in the two horizontal rows of the PC_{comp} code. The horizontal and global parities in the third PC_{fast} code is computed from the those of the old PC_{comp} code and those newly computed ones of the first two PC_{fast} codes. This optimization saves on the network transfers of two horizontal rows of data blocks. Similar to the upcode operation, data and vertical parity blocks in the resulting three PC_{fast} codes remain unchanged from the PC_{comp} code and do not require any network transfers.

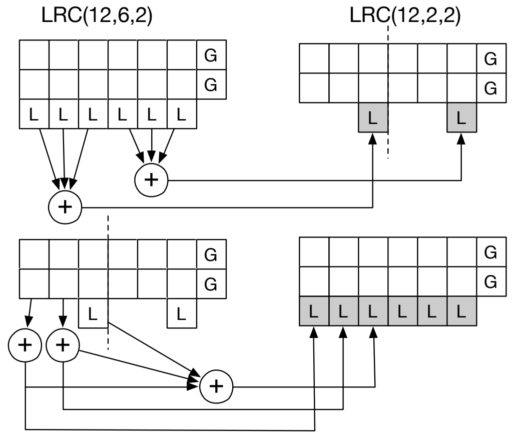


Figure 8: **LRC Code - Upcode and Downcode Operations:** *The figure shows the upcode and downcode operations with data and parity blocks for LRC codes. The shaded blocks are only computed during these operations, and remaining blocks remain unchanged.*

3.3 LRC Codes

We now describe the construction and coding interfaces of the erasure coding module using LRC codes in HACFS.

Encoding and Decoding. Figure 8 shows the construction of the LRC_{fast} or $LRC(12,6,2)$, with twelve data blocks, six local parities, and two global parities. The encode operation for an LRC code computes the local parities by performing an XOR over a *group* of data blocks. Two data blocks in each column form a different group in LRC_{fast}. The two global parities are computed by performing a Reed-Solomon encoding over all of the twelve data blocks [24]. The Reed-Solomon encoding of the global parities has properties similar to the LRC code construct used in Microsoft Azure Storage [15] for the most prominent single block failure scenarios. The decode operation for LRC_{fast} code is similar to Product Codes for data and local parity blocks. Any single failure in data or local parity blocks for LRC_{fast} requires two block transfers from the same column to reconstruct it. However, a failure in a global parity block requires all twelve data blocks to reconstruct it using the Reed-Solomon decoding.

Degraded reads from an HDFS client only occur on data blocks, while reconstructing a failed disk or node can also require recovering lost global parity blocks. As a result, the degraded read cost for the *fast code* - LRC_{fast} or $LRC(12,6,2)$ - is very low at two blocks (see Table 2). Unlike Product codes, the average reconstruction cost for the LRC_{fast} code is *asymmetri-*

cal to its degraded read cost since reconstruction requires twelve block transfers for global parity failures: $\frac{(12+6)*2+2*12}{12+6+2}$ or 3.25 blocks. However, the storage overhead for an LRC_{fast} code is 1.66x corresponding to eight parity blocks required for twelve data blocks.

We use the LRC_{comp} or $LRC(12,2,2)$ code used in Azure [15] as the *compact code* for adaptive coding in HACFS. The LRC_{comp} code has a lower storage overhead of 1.33x due to fewer local parities. However, each of its two local parities is associated with a group of six data blocks. Thus, recovering a lost data block or local parity requires six block transfers from its group in the LRC_{comp} code. The global parities require twelve data block transfers for recovery. As a result, the LRC_{comp} code also has a lower recovery cost for degraded reads than its reconstruction cost similar to the LRC_{fast} code. Both LRC codes are more reliable than three-way replication. We select a storage bound of 1.4x for the HACFS system with LRC codes since it is close to the storage overhead of LRC_{comp} code and lower than HACFS with Product codes.

Upcoding and Downcoding. Upcode and downcode operations for Product codes require merging three PC_{fast} codes into a PC_{comp} code and splitting a PC_{comp} code into three PC_{fast} codes respectively. The LRC codes can be upcoded and downcoded in a similar manner. However, such upcoding and downcoding with LRC codes requires several data block transfers to compute the new local and global parities. As a result, we use a more efficient code collapsing technique for the LRC upcode and downcode operations. This does not require computing the global parities again because collapsing converts exactly one LRC_{fast} code to one LRC_{comp} code (and reverse for downcoding).

Figure 8 shows the LRC upcode operation by computing the new local parities in LRC_{comp} code and preserving the global parities from the LRC_{fast} code. The two local parities in the LRC_{comp} code are computed as an XOR over three local parities in the LRC_{fast} code. As a result, the HACFS system requires only six network transfers to compute the two new local parities of the LRC_{comp} code in an upcode operation. The downcode operation computes two of the three new local parities in LRC_{fast} code from the data blocks in the individual columns of the LRC_{comp} code. The third local parity is computed by performing an XOR over the two new local parities and the old local parity in the LRC_{comp} code. Overall, the downcode operation requires ten block transfers for computing the new local parities. The global parities remain unchanged and do not require any network transfers in the downcode operation as well.

4 Implementation

We have implemented HACFS as an extension to the HDFS-RAID [3] module in the Hadoop Distributed File System (HDFS). The HDFS-RAID module is implemented by Facebook to support a single erasure code for distributed storage in an HDFS cluster. Our implementation of HACFS spans nearly 2 K lines of code, contained within the HDFS-RAID module, and requires no modification to other HDFS components such as the NameNode or DataNode.

Erasure Coding in HDFS. The HDFS-RAID module overlays erasure coding on top of HDFS and runs as a RaidNode process. The RaidNode process periodically queries the NameNode for new data files that need to be encoded and for corrupted files that need to be recovered. The RaidNode launches a MapReduce job to compute the parity files associated with data files on different DataNodes for the encode operation. The decode operation is invoked as part of a degraded read or the reconstruction phase.

A read from an HDFS client requests the block contents from a DataNode. A degraded read can occur due to failures on DataNodes such as a CRC check error. In those cases, the HDFS client queries the RaidNode for locations of the available blocks in the erasure-code stripe required for recovery. The client then retrieves these blocks and performs decoding itself to recover the failed block. The recovered block is used to serve the read request, but it is not written back to HDFS since most degraded reads are caused by transient failures that do not necessarily indicate data loss [20, 24].

When a disk or node failure is detected, the NameNode updates the list of corrupted blocks and lost files. The RaidNode then launches two MapReduce reconstruction jobs, one to recover lost data blocks and the other for lost parity blocks. The reconstruction job retrieves the available blocks for decoding, recovers the lost blocks using the decode operation, writes the recovered blocks to HDFS, and informs the NameNode of successful recovery. If there is a file which has many errors and can not be recovered, then it is marked as permanently lost.

HACFS and Two Erasure Codes. Figure 5 shows the three major components of HACFS implementation: erasure coding module, system states and the adaptive coding module. In addition, we also implement a fault injector to trigger degraded reads and data reconstruction.

The HDFS-RAID only supports a single Reed-Solomon erasure code for encoding data. We implement two new code families as part of the HACFS erasure coding module: Product and LRC codes. The erasure coding module in HACFS exports the same encode/decode interfaces as HDFS-RAID. In addition, the erasure coding module also provides two new upcode/downcode interfaces to the extended state machine implemented in the adaptive coding module of HACFS. The upcode operation either merges three fast codes for Product codes or collapses one fast code for LRC codes into a new compact code of smaller size. Downcoding performs the reverse sequence of steps. Both operations change the coding state of the data file and reduce its replication level to one.

The adaptive coding module tracks the system states and invokes the different coding interfaces. As desribed earlier, the HDFS-RAID module selects the three-way replicated files which are write cold based on their last modification time for encoding. The extended state machine implemented as part of the adaptive coding module in HACFS further examines these candidate files based on their read counts. It retrieves the coding state of all files classified as read cold and launches MapReduce jobs to upcode them into the compact code. Similarly, if the global system storage exceeds the bound, it upcodes the files with the lowest read counts into the compact code. If the global system storage is lower than the bound or a cold file has been accessed, the adaptive coding module downcodes the file into the fast code and also updates its coding state.

On a disk or node failure, the RaidNode in HACFS launches MapReduce jobs to recover lost data and parity blocks similar to HDFS-RAID. It prioritizes the jobs for reconstructing data over parities to quickly restore data availability for HDFS clients. There are four different types of reconstruction jobs in HACFS, which recover data and parity files encoded with fast and compact codes. Data files encoded with a fast code have a lower recovery cost, but they are also fewer in number than compact-coded data files. As a result, the reconstruction of data files encoded with a fast code is prioritized first among all reconstruction jobs. This prioritization also helps to reduce the total number of degraded reads during the reconstruction phase since fast-coded files get accessed more frequently.

We also implement a fault injector outside HDFS to simulate different modes of block, disk and node failures on DataNodes. The fault injector deletes a block from the local file system on a DataNode, which is detected by the HDFS DataNode as a missing block, and triggers a degraded read when an HDFS client tries to access it. The fault injector simulates a disk failure by deleting all data on a given disk of the target DataNode and then restarting the corresponding DataNode process. A node failure is injected by killing the target DataNode process itself. In both disk and node failure, the NameNode updates the list of lost blocks, and then the RaidNode launches the MapReduce jobs for reconstruction.

5 Evaluation

We evaluate HACFS's design techniques along three different axes: degraded read latency, reconstruction time and storage overhead.

5.1 Methods

Experiments were performed on a cluster of eleven different nodes, each of which is equipped with 24 Intel Xeon E5645 CPU cores running at 2.4 GHz, six 7.2 K RPM disks each of 2 TB capacity, 96 GB of memory, and 1 Gbps network link. The systems run Red Hat Enterprise Linux 6.5 and HDFS-RAID [3]. We use the default HDFS filesystem block size of 64 MB.

The HACFS system uses adaptive coding with fast and compact codes from Product and LRC code families. We refer these two different systems as: HACFS-PC using PC_{fast} and PC_{comp} codes, and HACFS-LRC using LRC_{fast} and LRC_{comp} codes. We compare these two HACFS systems against three HDFS-RAID systems using exactly one of these codes for erasure coding: Reed-Solomon $RS(6,3)$ code, Reed-Solomon $RS(10,4)$ code, and $LRC(12,2,2)$ or LRC_{comp} code. These three codes are used in production storage systems: $RS(6,3)$ used in Google Colossus FS [2], $RS(10,4)$ used in Facebook HDFS-RAID [3], and LRC_{comp} used in Microsoft Azure Storage [15]. We configure the storage overhead bound for HACFS-PC and HACFS-LRC systems as 1.5x (similar to Colossus FS) and 1.4x (similar to Facebook's HDFS-RAID) respectively.

We use the default HDFS-RAID block placement scheme to evenly distribute data across the cluster ensuring that no two blocks within an erasure code stripe reside on the same disk. We measure the degraded read latency by injecting single block failures (as described in Section 4) for a MapReduce grep job that is both network and I/O intensive. We measure the reconstruction time by deleting all blocks on a disk. The block placement scheme ensures that the lost disk does not have two blocks from the same stripe. As a result, the NameNode starts the reconstruction jobs in parallel using the remaining available disks. We report the completion time and network bytes transferred for reconstruction jobs averaged over five different executions.

We use five different workloads collected from production Hadoop clusters in Facebook and four Cloudera customers [9]. Table 3 shows the distribution of each workload: number of files accessed, percentage of files accounting for 90% of the total accesses, percentage of data volume corresponding to such files, and percentage of reads in all accesses to such files.

5.2 System Comparison

HACFS improves degraded read latency, reconstruction

	HACFS-PC			HACFS-LRC		
	Colossus FS	HDFS-RAID	Azure	Colossus FS	HDFS-RAID	Azure
Degraded Read Latency	25.2%	46.1%	25.4%	21.5%	43.3%	21.2%
Reconstruction Time	14.3%	43.7%	21.4%	-3.1%	32.2%	5.6%
Storage Overhead	2.3%	-4.7%	-10.2%	7.7%	1.1%	-4.2%

Table 4: **System Comparison**. The table shows the percentage improvement of two HACFS systems using Product and LRC codes for recovery performance and storage overhead over three single code systems: Google ColossusFS using $RS(6,3)$, Facebook HDFS-RAID using $RS(10,4)$, and Microsoft Azure Storage using LRC_{comp} codes.

Workload	Files	Hot Files	% Hot Data	% Hot Reads
CC1	20.1K	1.2K	5.9	86.1
CC2	10.2K	1.6K	15.7	75.9
CC3	2.1K	1.1K	52.4	75.5
CC4	5.2K	1.4K	26.9	85.2
FB	802K	103K	12.8	90.2

Table 3: **Workload Characteristics**: The table shows the distributions of five different workloads from Hadoop clusters deployed at four different Cloudera customers (CC1/2/3/4) and Facebook (FB).

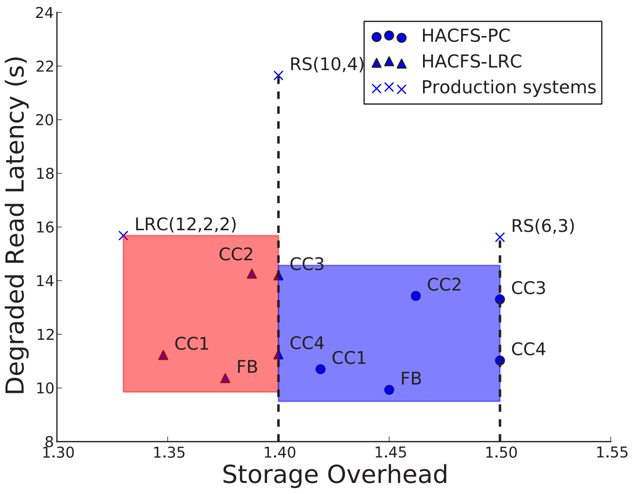


Figure 9: **Degraded Read Latency:** The figure shows the degraded read latency and storage overhead for two HACFS systems and three single code systems.

time, and provides low and bounded storage overhead. We begin with a high-level comparison of HACFS using adaptive coding on Product and LRC codes with three single code systems: ColossusFS, HDFS-RAID and Azure.

Degraded Read Latency. Table 4 shows the percentage improvement of adaptive coding in HACFS with Product and LRC codes averaged over five different workloads. HACFS reduces the degraded latency by 25-46% for Product codes and 21-43% for LRC codes compared to three single-coded systems. This improvement in HACFS primarily comes from the use of fast codes (PC_{fast} and LRC_{fast}) for hot data, which is primarily dominated by read accesses (see Table 3). As a result, the degraded read latency of HACFS is lower than all of the three other production systems relying on $RS(6,3)$, $RS(10,4)$ and LRC_{comp} codes. We describe these results in more detail for each of the five different workloads in Section 5.3.

Reconstruction Time. HACFS improves the reconstruction time to recover from a disk or node failure by 14-43% for Product codes and up to 32% for LRC codes. The reconstruction time is dominated by the volume of data and parity blocks lost in a disk or node failure. The fast and compact Product codes used in HACFS have a lower reconstruction cost than the two LRC codes. As described in Section 3.3, this is because LRC codes have a higher recovery cost for failures in local and global parity blocks than data blocks. As a result, the HACFS system with LRC codes takes slightly longer to reconstruct a lost disk than ColossusFS, which uses the $RS(6,3)$ code with a symmetric cost to recover from data and parity failures. We discuss these results in more detail in Section 5.4.

Storage Overhead. HACFS is designed to provide low and bounded storage overheads. The Azure system using the LRC_{comp} code has the lowest storage overhead (see Table 2), and is up to 4-10% better than the two HACFS systems. The HDFS-RAID system using $RS(10,4)$ has about 5% lower storage overhead than HACFS optimized for recovery with Product codes. However, the HACFS system with LRC codes has storage overheads lower or comparable to the three single-coded production systems [2, 15, 24]. This is primarily because adaptive coding in HACFS bounds the storage overhead by 1.5x for Product codes and by 1.4x for LRC codes. We discuss the storage overheads of each system across different workloads in Section 5.3.

5.3 Degraded Read Latency

HACFS uses a combination of recovery-efficient fast codes (PC_{fast} and LRC_{fast}) and storage-efficient compact codes (PC_{comp} and LRC_{comp}). Figure 9 shows the degraded read latency on y-axis and storage overhead on x-axis for the five different workloads. A normal read from an HDFS client to an available data block can take up to 1.2 seconds since it requires one local

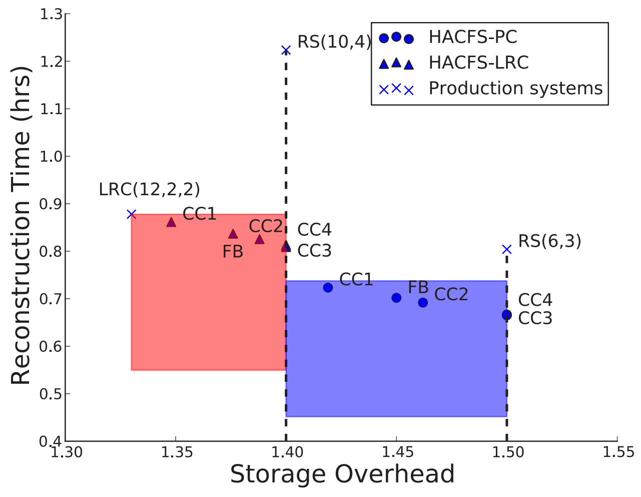


Figure 10: **Reconstruction Time:** The figure shows the reconstruction time to recover from data loss with two HACFS systems and three single code systems.

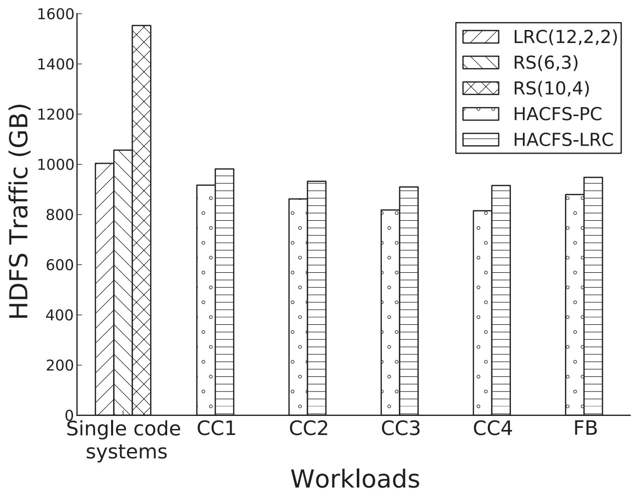


Figure 11: **Reconstruction Traffic:** The figure shows the reconstruction traffic for recovering lost data and parity blocks with two HACFS systems and three single code systems.

disk read and one network transfer if the block is remote. In contrast, a degraded read can require multiple network transfers, and takes between 16-21 seconds for the three single coded systems. These systems do not adapt with the workload and only use a single code. As a result, their degraded read latency and storage overhead is the same across all five workloads. Adaptive coding in HACFS reduces the degraded read latency by 5-10 seconds for three workloads (CC1, CC4 and FB), which have a higher percentage of reads to hot data encoded with the fast code (85-90%, see Table 3). The two shaded boxes in Figure 9 demonstrate that HACFS adapts to the characteristics of the different workloads. However, HACFS always outperforms the three single coded systems since all of them require more blocks to be read and transferred over the network to decode a missing block.

Both HACFS systems have a lower storage overhead for workloads (CC1, CC2 and FB) with a higher percentage of cold files (85-95%) encoded with the compact codes. The lowest possible storage overhead for HACFS is shown by the left boundary of the two shaded regions marked with 1.33x and 1.4x for the compact codes (LRC_{comp} and PC_{comp} codes respectively). In addition, HACFS also bounds the storage overhead by 1.5x for Product codes and 1.4x for LRC codes. As a result, workloads (CC3 and CC4) with fewer cold files still never exceed these storage overheads marked by the right edges in the two shaded regions. If we do not enforce any storage overhead bounds, these two workloads benefit even further by a reduction of 6-20% in their degraded read latencies.

5.4 Reconstruction Time

Figure 10 shows the reconstruction time of the three single code systems and HACFS system with adaptive coding when a disk with 100 GB of data fails. The reconstruction job launches map tasks on different DataNodes to recreate the data and parity blocks from the lost disk. The time to reconstruct a cold file encoded with a compact code is longer than that for a fast code. The HACFS system with Product codes outperforms the three single code systems for all five workloads. It takes about 10-35 minutes less reconstruction time than the three single code systems. This is because both fast and compact Product codes reconstruct faster than the two Reed-Solomon codes and the LRC_{comp} code.

The HACFS system with the use of faster LRC_{fast} code for reconstruction outperforms the LRC_{comp} code with the lowest storage overhead. However, the HACFS system with LRC codes is generally worse for all workloads than the $RS(6,3)$ code used in the ColossusFS. This is because both LRC codes used in HACFS have a recovery cost for global parities higher than the $RS(6,3)$ code (see Table 2).

Figure 11 shows the reconstruction traffic measured as HDFS read and writes incurred by the reconstruction job to recover 100 GB of data and additional parity blocks lost from the failed disk. The reconstruction job reads all blocks in the code stripe for recovering the lost blocks, and then writes the recovered data and parity blocks back to HDFS. The HDFS-RAID system using the $RS(10,4)$ code results in the highest traffic: 1550 GB of reconstruction traffic for 100 GB of lost data. This is close to the theoretical reconstruction traffic of nearly fifteen blocks per lost data block, including ten block reads for

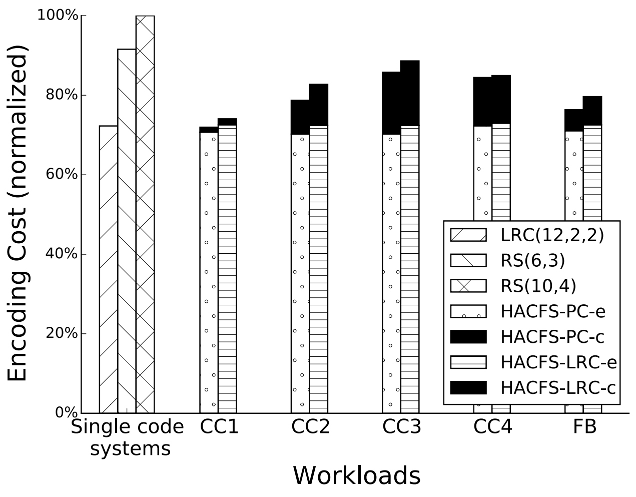


Figure 12: **Encoding Cost:** The figure shows the encoding time for three single code systems and two HACFS systems normalized over the HDFS-RAID ($RS(10,4)$). The black bars show the *c*-onversion time component of the total *e*-ncoding time for the two HACFS systems.

data recovery, four block reads for parity recovery, and then writes of the recovered data block and parity blocks ($RS(10,4)$ uniformly stores 0.4 parity blocks with each data block on a disk). Similarly, LRC_{comp} in Azure and $RS(6,3)$ in Colossus, require nearly ten HDFS block read/writes for recovering a block from the lost disk.

The HACFS system with Product codes always requires fewer blocks for reconstruction than the three single code systems: between eight blocks for CC3 and CC4 workloads and nine blocks for CC1, CC2 and FB workloads (with more than 85% cold files) based on the data skew distributions. The HACFS system with LRC codes requires more blocks for global parity recovery than Product codes. As a result, its reconstruction traffic is close to $RS(6,3)$ and LRC_{comp} codes at nearly ten blocks per lost data block.

5.5 Encoding and Conversion Time

Figure 12 shows the encoding cost for initial encoding of three-way replicated data in three single code systems, and the encoding cost for initial encoding and later conversion between the fast and compact codes in the two HACFS systems. We normalize the encoding cost per block to eliminate the differences in dataset sizes across the five workloads. All compared systems are based on HDFS-RAID implementation, which schedules the encoding and conversion operations as MapReduce jobs in background to minimize their impacts on user jobs. As a result, we show the impact of encoding cost for all systems relative to $RS(10,4)$ used in HDFS-RAID in Figure 12. Encoding cost is a function of the coding scheme used for data blocks and does not change with workload for the three single code systems. Reed-Solomon codes used in HDFS-RAID and ColossusFS have the highest encoding cost because of complex Galois Field operations required to compute parity blocks [18]. LRC code in Azure uses such operations only to compute global parities and uses cheaper XOR operations for all local parities. Similarly, HACFS with Product codes only uses XOR operations for encoding. As a result, the encoding time component of the two HACFS systems is similar to the LRC codes in Azure for all workloads.

The HACFS system also converts (upcodes/downcodes) data between fast and compact codes. The conversion cost is only high when the HACFS system aggressively converts blocks to limit the storage overhead by upcoding hot files into compact code. As a result, the three workloads (CC2, CC3 and CC4) with a higher percentage of hot data spend up to 18% of total encoding time for conversion operations. For these workloads, the total encoding and conversion cost of the HACFS systems is up to 16% higher than the Azure system using a single LRC code. In general, the encoding cost of the two HACFS systems is about 3-28% lower than the single-code ColossusFS and HDFS-RAID systems using Reed-Solomon codes for all workloads.

6 Related Work

Our work builds on past work on distributed storage systems, faster recovery techniques, and tiered storage systems.

Distributed Storage Systems. Petabytes of storage is becoming common with the fast growing data requirements of modern systems today. Erasure codes offer an attractive alternative to provide lower storage overhead than data replication. As a result, many distributed filesystems such as Google ColossusFS [2], Facebook HDFS [3], and IBM General Parallel File System [6] are moving to the use of erasure codes. Many popular object stores used for cloud storage, for example, OpenStack Swift [17], Microsoft Azure Storage [15] and Cleversafe [1] are also adopting erasure codes for low storage overhead. However, most of these systems use a single erasure code and address the recovery cost by trading for storage overhead. In contrast, HACFS is the first system that uses a combination of two codes to dynamically adapt with workload changes and provide both low recovery cost and storage overhead.

Faster Recovery for Erasure-Codes. Recently, there has been a growing focus on improving recovery performance for erasure-coded storage systems. Reed-Solomon codes [21] offer optimal storage overhead but generally have high recovery cost. Rotated Reed-Solomon codes have been proposed as an alternative construction, which requires fewer data reads for faster degraded read recovery [16]. HitchHiker proposes a new encoding technique by dividing a single Reed-Solomon code stripe into two correlated substripes and improves recovery performance [20]. However, both of them trade extra encoding time for faster recovery. In contrast, adaptive coding techniques in HACFS provide lower recovery cost without increasing encoding time. In general, adaptive coding can be applied to most code families, which tradeoff between storage overhead and recovery cost. We have found efficient up/downcode operations for applying adaptive coding to different constructs of Reed-Solomon code and other modern storage codes such as PMDS [8] and HoVer [12]. For example, we devised up/downcode operations for converting m (n,r) Reed-Solomon codes into a (mn, r) Reed-Solomon code using a parity-only conversion scheme.

Tiered Storage Systems. Adaptive coding in HACFS is inspired by tiering in RAID architectures [27, 3, 10]. AutoRAID [27] provides a two-level storage hierarchy within the storage controller. It automatically migrates data between different RAID levels to provide high I/O performance for active data and low storage overhead for inactive data. Similarly, HACFS migrates data between fast and compact erasure codes, however with the objective to reduce extra network transfers for recovery in distributed storage. Facebook's HDFS-RAID [3] and DiskReduce [10] propose tiered storage by asynchronously migrating data between replicated and erasure-coded storage tiers. HACFS extends this further by splitting the erasure-coded storage tier into two parts to optimize for both storage overhead and recovery performance.

7 Conclusions

Distributed storage systems extensively deploy erasure-coding today for lower storage overhead than data replication. However, most of these systems trade storage overhead for recovery performance. We present a novel erasure-coded storage system, which uses two different erasure codes and dynamically adapts by converting between them based on workload characteristics. It uses a fast code for fast recovery performance and a compact code for low storage overhead. In the future, as we move to cloud storage, it will be important to revisit similar erasure-coding tradeoffs, and extend adaptive coding techniques to large-scale object stores in the cloud.

Acknowledgements

We thank Jim Hafner for his valuable feedback at different stages of this work. We also thank our shepherd Cheng Huang and the anonymous FAST reviewers for their helpful comments and useful feedback.

References

[1] Cleversafe object storage. `http://www.cleversafe.com`.

[2] Colossus, successor to Google File System. `http://static.googleusercontent.com/media/research.google.com/en/us/university/relations/facultysummit2010/storage_architecture_and_challenges.pdf`.

[3] Facebook's erasure coded hadoop distributed file system (HDFS-RAID). `https://github.com/facebook/hadoop-20`.

[4] The Hadoop Distributed File System. `http://wiki.apache.org/hadoop/HDFS`, 2007.

[5] Cloudera: What do real-life apache hadoop workloads look like? `http://blog.cloudera.com/blog/2012/09/what-do-real-life-hadoop-workloads-look-like/`, September 2012.

[6] An introduction to GPFS version 3.5. `http://www-03.ibm.com/systems/resources/introduction-to-gpfs-3-5.pdf`, August 2012.

[7] ABAD, C. L., ROBERTS, N., LU, Y., AND CAMPBELL, R. H. A storage-centric analysis of mapreduce workloads: File popularity, temporal locality and arrival patterns. In *Proceedings of the 2012 IEEE International Symposium on Workload Characterization (IISWC)* (Washington, DC, USA, 2012), IISWC '12, IEEE Computer Society, pp. 100–109.

[8] BLAUM, M., HAFNER, J. L., AND HETZLER, S. Partial-MDS codes and their application to RAID type of architectures. *IEEE Transactions on Information Theory 59*, 7 (2013), 4510–4519.

[9] CHEN, Y., ALSPAUGH, S., AND KATZ, R. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *Proc. VLDB Endow. 5*, 12 (Aug. 2012), 1802–1813.

[10] FAN, B., TANTISIRIROJ, W., XIAO, L., AND GIBSON, G. DiskReduce: RAID for data-intensive scalable computing. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage* (New York, NY, USA, 2009), PDSW '09, ACM, pp. 6–10.

[11] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 29–43.

[12] HAFNER, J. L. HoVer erasure codes for disk arrays. In *Proceedings of the International Conference on Dependable Systems and Networks* (Washington, DC, USA, 2006), DSN '06, IEEE Computer Society, pp. 217–226.

[13] HAFNER, J. L., AND RAO, K. Notes on reliability models for non-MDS erasure codes. IBM Tech Report RJ10391 (A0610-035), 2006.

[14] HUANG, C., CHEN, M., AND LI, J. Pyramid codes: Flexible schemes to trade space for access efficiency in reliable data storage systems. *Trans. Storage 9*, 1 (Mar. 2013), 3:1–3:28.

[15] HUANG, C., SIMITCI, H., XU, Y., OGUS, A., CALDER, B., GOPALAN, P., LI, J., AND YEKHANIN, S. Erasure coding in Windows Azure Storage. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (2012), USENIX ATC'12, USENIX Association, pp. 2–2.

[16] KHAN, O., BURNS, R., PLANK, J., PIERCE, W., AND HUANG, C. Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (2012), FAST'12, USENIX Association, pp. 20–20.

[17] LUSE, P., AND GREENAN, K. Swift object storage: Adding erasure codes. `http://www.snia.org/sites/default/files/Luse_Kevin_SNIATutorialSwift_Object_Storage2014_final.pdf`, 2014.

[18] PLANK, J. S., GREENAN, K. M., AND MILLER, E. L. Screaming fast Galois Field arithmetic using Intel SIMD instructions. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies* (2013), FAST'13, USENIX Association, pp. 299–306.

[19] RASHMI, K. V., SHAH, N. B., GU, D., KUANG, H., BORTHAKUR, D., AND RAMCHANDRAN, K. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster. In *Proceedings of the 5th USENIX Conference on Hot Topics in Storage and File Systems* (2013), HotStorage'13, USENIX Association, pp. 8–8.

[20] RASHMI, K. V., SHAH, N. B., GU, D., KUANG, H., BORTHAKUR, D., AND RAMCHANDRAN, K. A "hitchhiker's" guide to fast and efficient data reconstruction in erasure-coded data centers. In *Proceedings of ACM SIGCOMM* (2014), SIGCOMM'14.

[21] REED, I. S., AND SOLOMON, G. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics 8*, 2 (1960), 300–304.

[22] REN, K., KWON, Y., BALAZINSKA, M., AND HOWE, B. Hadoop's adolescence: An analysis of Hadoop usage in scientific workloads. *Proc. VLDB Endow. 6*, 10 (Aug. 2013), 853–864.

[23] ROTH, R. *Introduction to Coding Theory*. Cambridge University Press, New York, NY, USA, 2006.

[24] SATHIAMOORTHY, M., ASTERIS, M., PAPAILIOPOULOS, D., DIMAKIS, A. G., VADALI, R., CHEN, S., AND BORTHAKUR, D. XORing elephants: novel erasure codes for big data. In *Proceedings of the 39th international conference on Very Large Data Bases* (2013), PVLDB'13, VLDB Endowment, pp. 325–336.

[25] SCHMUCK, F., AND HASKIN, R. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies* (2002), FAST '02, USENIX Association.

[26] TAMO, I., AND BARG, A. A family of optimal locally recoverable codes. *IEEE Transactions on Information Theory 60*, 8 (2014), 4661–4676.

[27] WILKES, J., GOLDING, R., STAELIN, C., AND SULLIVAN, T. The HP AutoRAID hierarchical storage system. *ACM Trans. Comput. Syst. 14*, 1 (Feb. 1996), 108–136.

How Much Can Data Compressibility Help to Improve NAND Flash Memory Lifetime?

Jiangpeng Li[*], Kai Zhao[*], Xuebin Zhang[*], Jun Ma[†], Ming Zhao[‡]and Tong Zhang[*]
[*]*ECSE department, Rensselaer Polytechnic Institute, USA*
[†]*Shanghai Jiao Tong University, China*
[‡]*School of Computing and Information Sciences, Florida International University, USA*
{*lijiangpeng1984@gmail.com, tzhang@ecse.rpi.edu*}

Abstract

Although data compression can benefit flash memory lifetime, little work has been done to rigorously study the full potential of exploiting data compressibility to improve memory lifetime. This work attempts to fill this missing link. Motivated by the fact that memory cell damage strongly depends on the data content being stored, we first propose an implicit data compression approach (i.e., compress each data sector but do not increase the number of sectors per flash memory page) as a complement to conventional explicit data compression that aims to increase the number of sectors per flash memory page. Due to the runtime variation of data compressibility, each flash memory page almost always contains some unused storage space left by compressed data sectors. We develop a set of design strategies for exploiting such unused storage space to reduce the overall memory physical damage. We derive a set of mathematical formulations that can quantitatively estimate flash memory physical damage reduction gained by the proposed design strategies for both explicit and implicit data compression. Using 20nm MLC NAND flash memory chips, we carry out extensive experiments to quantify the content dependency of memory cell damage, based upon which we empirically evaluate and compare the effectiveness of the proposed design strategies under a wide spectrum of data compressibility characteristics.

1 Introduction

NAND flash memory cells gradually wear out with program/erase (P/E) cycling due to physical device damage caused by each P/E cycle, and cycling endurance drastically degrades with the technology scaling down. Hence, how to maximize memory lifetime has been widely studied from different aspects, e.g., signal processing and error correction coding (ECC) [1–3], flash translation layer (FTL) [4–10], and system software stack [11–13].

Nevertheless, to our best knowledge, no prior work has thoroughly studied how data compressibility can be leveraged to improve flash memory lifetime. It is actually not surprising, since this question appears to be trivial at first glance: In conventional practice, the sole objective of data compression is to improve storage efficiency (i.e., explicitly increase the number of data sectors that can be stored in one flash memory page). This is referred to as *explicit data compression* in this work. Due to the runtime variation of data compressibility, explicit data compression results in heterogeneity among flash memory pages in terms of the number of sectors per page, which can complicate FTL and/or file system design. As a result, it is not uncommon that commercial flash-based storage devices do not use data compression at all. If sophisticated FTL and/or file systems, which can employ explicit compression to improve storage efficiency, are indeed available, one may simply expect that storing data with an average compression ratio[1] of α can directly improve the flash memory lifetime by $1/\alpha$. Therefore, one may easily draw the following conclusion: If we do not want to complicate the FTL and/or file system, we should simply leave the user data uncompressed, for which the data compressibility is totally irrelevant to flash memory lifetime; If we use complicated FTL and/or file systems to support explicit data compression, the flash memory lifetime improvement solely depends on the average data compression ratio.

This work contends that the above intuitive conclusion is far from revealing the complete potential of how data compressibility can help to improve flash memory lifetime. In essence, it overlooks two factors. First, flash memory experiences *content-dependent memory damage*, i.e., the damage suffered by each memory cell depends on its content (e.g., '11', '10', '00', and '01' in MLC flash memory) being stored. Once data compression leaves some unused storage space within flash

[1]Let S_o and S_c denote the size of the original and compressed data, then we define *compression ratio* as S_c/S_o, which falls into $(0,1]$.

memory pages, we can manipulate their data content in a *damage-friendly* manner to reduce physical damage. Hence, conventional explicit data compression is not necessarily the only option of exploiting data compressibility to improve memory lifetime. We propose *implicit data compression* as an alternative to complement with explicit data compression. With implicit data compression, we compress each data sector but do not increase the number of data sectors per flash memory page. Therefore, implicit compression has no impact on FTL and/or file system but meanwhile does not improve storage efficiency either. Second, for multi-bit per cell (e.g., MLC and TLC) flash memory, physical damage depends on a variety of factors (e.g., distribution characteristics of compressed data size, relative placement or layout of different pages on the same memory wordline), which have not been considered in prior work.

This paper presents a thorough study on exploiting data compressibility to reduce physical damage and hence improve flash memory lifetime. Since random read latency is one of the most important metrics of flash-based storage devices, this work assumes that each compressed data sector must reside entirely in one flash memory page. As a result, each flash memory page almost always contains some unused storage space left by compressed data sectors. Motivated by the content dependency of flash memory cell damage, we present a set of design strategies that can exploit the unused storage space within a flash memory page to reduce the overall memory damage, for both explicit and implicit data compression. Then we derive a set of mathematical formulations for quantitatively estimating flash memory damage reduction gained by the proposed design strategies. These rigorous mathematical formulations build a framework that directly links flash memory lifetime with data compressibility characteristics (e.g., mean and deviation of data compression ratio) and memory cell damage content dependency. Using 20nm MLC NAND flash memory chips, we carried out experiments to quantitatively measure the content-dependent memory cell damage factors, based upon which we empirically evaluated and compared the effectiveness of the proposed design strategies with either explicit or implicit compression. In summary, the main contributions of this work include:

1. We propose an implicit data compression strategy as a viable complement to conventional explicit data compression for exploiting data compressibility to improve flash memory lifetime;

2. A set of design strategies are developed to leverage the unused storage space left by data compression within flash memory pages to reduce the memory cell physical damage;

3. We derive a set of mathematical formulations to accurately estimate the flash memory damage based upon the characteristics of data compressibility and content-dependent memory cell damage;

4. We quantitatively compare explicit data compression and implicit data compression under a wide spectrum of runtime data compressibility characteristics and show that it is important to fully understand the data compressibility characteristics in order to choose the appropriate design strategy.

Finally, we note that, although this work focuses on flash memory, the developed design strategies and mathematical formulations are readily applicable to other emerging memory technologies, e.g., PCM and ReRAM, that experience similar content dependency of memory cell physical damage.

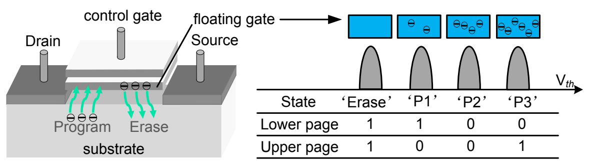
2 Design Strategies

This section presents a set of design strategies that can exploit data compressibility to reduce memory cell damage. We first discuss the content dependency of cycling-induced memory damage that motivates us to propose implicit compression (i.e., compress the data without increasing the number of sectors per flash memory page) in addition to the conventional explicit compression (i.e., compress the data and increase the number of sectors per page as much as possible). We further present different strategies on laying out the compressed data within flash memory pages that aim to leverage the content-dependent memory damage phenomenon for improving flash memory lifetime. We note that this work only focuses on MLC memory, and the discussions could be readily extended to the more complicated TLC case.

2.1 Content-Dependent Damage

NAND flash memory handles data programming and read in page units with a typical size of 4kB or 8kB. For high-density MLC and TLC memory, different bits within each MLC/TLC memory cell belong to different pages. This can be illustrated in Fig. 1 for MLC flash memory, where the two bits within each memory cell belong to lower and upper pages, respectively.

NAND flash memory cells wear out with P/E cycling due to the oxide damage caused by the electrons that pass through the gate oxide during each P/E cycle. Although current practice estimates the memory cell damage solely dependent upon the number of P/E cycles endured by memory cells, actual physical damage further depends on the data content being programmed to memory cells.



State	'Erase'	'P1'	'P2'	'P3'
Lower page	1	1	0	0
Upper page	1	0	0	1

Figure 1: Illustration of MLC NAND flash memory cell.

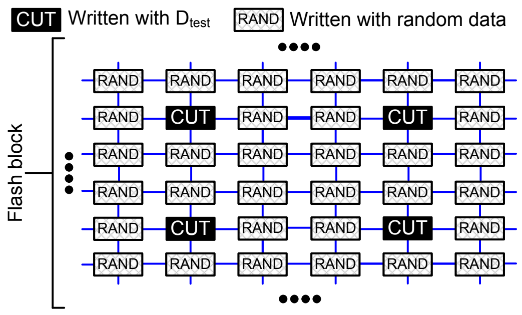


Figure 2: Illustration of method for accurately quantifying the flash memory cell damage caused by each distinct data content.

This can be intuitively explained using Fig. 1: different data content (e.g., '11', '10', '00', and '01') correspond to different number of electrons that pass through the gate oxide, and hence different amount of physical damage [14, 15].

To further demonstrate such content dependency, we carried out experiments using 20nm MLC NAND flash memory chips. To evaluate the effect of writing the content $D_{test} \in \{\text{'11'},\text{'10'},\text{'00'},\text{'01'}\}$, we program each flash memory block with the pattern as shown in Fig. 2, i.e., to examine the cell content D_{test}, each memory cell written with D_{test} is surrounded by memory cells written with random data. This can incorporate the effect of cell-to-cell interference and program disturb in practice. The memory cells written with D_{test} are called *cells under test* (CUT). Different memory blocks are used for testing different D_{test}, and the locations of all the CUTs are fixed throughout the entire cycling. We capture the raw bit error rate (BER) of all the CUTs every few hundreds cycles by writing random data to all the memory cells. The measurement results are shown in Fig. 3.

2.2 Data Storage Schemes

Since each compressed sector resides entirely in one memory page, each page will have a certain amount of unused storage space. This subsection first discusses how we should determine the content of the unused storage space to minimize the overall damage, then discusses different options of laying out the compressed data within flash memory pages.

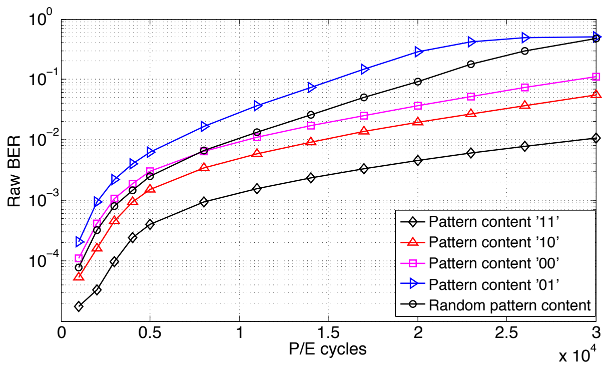


Figure 3: Measured memory raw bit error rate (BER) vs. cycling with different data content.

2.2.1 Content of Unused Storage Space

For MLC NAND flash memory, each pair of lower and upper pages together determine the memory cell content and hence the flash memory damage. To minimize the flash memory damage, we should appropriately determine the data content in the unused storage space left by data compression. Let $S^{(l)}$ and $S^{(u)}$ denote the unused storage space in lower and upper page, and b_l and b_u denote the two bits in the same memory cell and belong to lower and upper page, respectively. Recall that the memory cell damage caused by the content '11', '10', '00', and '01' monotonically increase (where the left bit and right bit resides in lower and upper page, respectively), as illustrated in Fig. 3. Therefore, for each memory cell, we should apply the following rules to minimize flash memory damage:

- If $b_l \in S^{(l)}$ and $b_u \in S^{(u)}$ (i.e., we can freely set the values of both bits), we set $b_l = b_u = 1$ hence the least harmful content '11' is written to the cell;
- If $b_l \in S^{(l)}$ and $b_u \notin S^{(u)}$ (i.e., we can only freely set the value of b_l), we always set b_l as '1' regardless to the value of b_u;
- If $b_l \notin S^{(l)}$ and $b_u \in S^{(u)}$ (i.e., we can only freely set the value of b_u), we always set $b_u = b_l$.

2.2.2 Compressed Data Layout

Since the memory cells covered by $S^{(l)}$ or $S^{(u)}$ experience less damage than the other memory cells, we should keep shifting the location of $S^{(l)}$ and $S^{(u)}$ within flash memory pages in order to equalize the damage among all the memory cells. We define a parameter l_{head} to represent the location from where the compressed data are continuously stored in the lower and upper pages. We should keep changing l_{head} in order to equalize the memory cell damage. Since the storage device FTL module

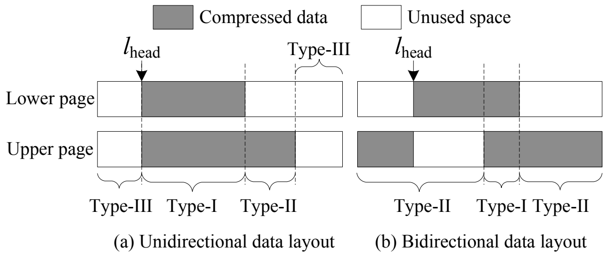


Figure 4: Two different data layout strategies.

always keeps track of the P/E cycles of each memory block, we can fix a relationship between l_{head} and P/E cycle number, e.g., let L denote the memory page size and $N_{P/E}$ denote the P/E cycle number, we can calculate $l_{head} = \lfloor \lfloor t \cdot N_{P/E} \rfloor \bmod L \rfloor$, where t is a fixed constant integer. As a result, the storage device controller does not need to record the value of l_{head} for each memory block. In addition, as decompression is a process that is done serially, the length of the compressed data need not be kept in the FTL. For each compressed memory page, the decompression process can be terminated once the decompressed data length reaches the page length. Therefore, in order to support the proposed design strategy, the only overhead at the FTL layer is to calculate the l_{head} for each memory page.

For MLC NAND flash memory, there are two different options for laying out the compressed data in lower and upper pages. As illustrated in Fig. 4, the first option is to lay out the compressed data towards the same direction from l_{head} in both the lower and upper pages, that we refer is referred to as *unidirectional data layout*. The other option is to lay out the compressed data towards opposite directions in the lower and upper pages, that we refer to as *bidirectional data layout*. As shown in Fig. 4, all the memory cells can be categorized into three types: (1) In each type-I memory cell, both bits belong to the compressed data; (2) In each type-II memory cell, one bit belongs to the compressed data while the other bit belongs to the unused storage space; (3) In each type-III memory cell, both bits belong to the unused storage space. Apparently, the physical damage experienced by type-I, type-II, and type-III memory cells monotonically reduces. Compared with unidirectional data layout, bidirectional data layout leads to more type-II memory cells and less type-I and type-III memory cells.

2.2.3 Conditional Data Exchange

According to the discussion in Section 2.2.1, the content of each type-II memory cell can only belong to {'11', '10'} or {'11', '00'} if the lower or upper page bit belongs to unused storage space. As shown in Fig. 3, '10' causes less damage than '00'. Hence, the memory damage tends to be less if the lower page has more unused storage space (i.e., data being stored in the lower page have better compressibility). This observation directly motivates us to propose *conditional data exchange*: Let $D^{(l)}$ and $D^{(u)}$ denote the compressed data that have been originally arranged by the storage device FTL to store in one pair of lower and upper pages. If the length of $D^{(l)}$ is not larger than that of $D^{(u)}$ (i.e., $|D^{(l)}| \leq |D^{(u)}|$), we directly store $D^{(l)}$ and $D^{(u)}$ to the lower and upper pages, respectively; otherwise we switch their page location, i.e., store $D^{(l)}$ to the upper page and $D^{(u)}$ to the lower page.

Although this design scheme can reduce flash memory damage, it could complicate the FTL design. If the FTL uses the page-level address mapping, we need to update the mapping table once the data exchange operation occurs. This will not introduce any mapping table storage overhead. If the FTL uses block-level or hybrid page/block-level address mapping, we must keep a 1-bit flag for each memory wordline, leading to extra mapping table storage overhead.

3 Mathematical Formulations

This section presents the mathematical formulations that can accurately estimate flash memory physical damage reduction when using the design strategies presented in Section 2, for both explicit and implicit data compression. It is evident that different types of data can have different compressibility characteristics. With the popular LZ77 [16] compression algorithm and sector size of 4kB, Fig. 5 shows the per-sector compression ratio distribution for some common types of data. The results show that the compression ratio tends to approximately follows a Gaussian distribution. We carried out further experiments to verify the accuracy of such distribution approximation. Fig. 5 shows the absolute difference (denoted as "Appr. error" in the figure) between the exact distribution and the approximate distribution for different types of data. The corresponding mean square errors (MSE) for these types of data are all at the magnitude of 10^{-5}. Therefore, we can conclude that such a Gaussian-based approximation is reasonable with almost negligible inaccuracy. Therefore, to facilitate the mathematical derivation, we set that per-sector data compression ratio follows a Gaussian distribution in this work.

3.1 Content-dependent Damage Factor

We first introduce a parameter, called normalized content-dependent damage factor, to quantify the impact of different content on memory cell damage. Let BER_{max} denote the maximum memory raw BER that can be tolerated by the storage device error correction mechanism.

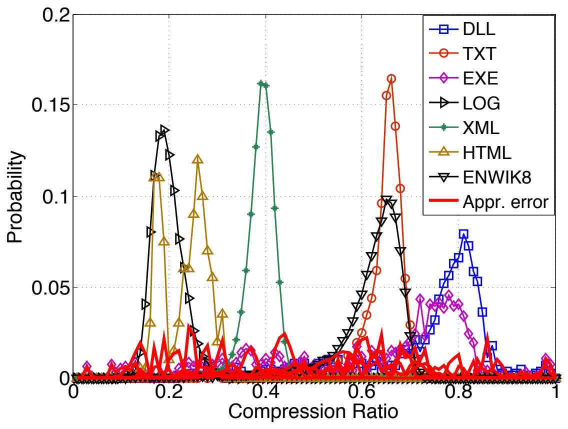


Figure 5: Measured distribution of compression ratio for different types of data.

For l-bit/cell NAND flash memory, let $\Psi^{(i)}(\eta)$ denote the raw BER after we keep programming memory cells with the same content $i \in [0, 2^l - 1]$ for η cycles. Let $\Psi^{(r)}(\eta)$ denote the raw BER after we have programmed memory cells with random content for η cycles. Let $\eta_{max}^{(i)}$ and $\eta_{max}^{(r)}$denote the P/E cycle number under which the raw BER $\Psi^{(i)}(\eta)$ and $\Psi^{(r)}(\eta)$ equal to BER_{max}, respectively. Hence, we can estimate that the physical memory cell damage caused by each programming operation with the content i is proportional to $1/\eta_{max}^{(i)}$. In addition, on average the physical memory cell damage caused by programming random content is proportional to $1/\eta_{max}^{(r)}$. In this work, we define the content-dependent damage factor ρ_i for each content i by normalizing with the average damage caused by random content, i.e.,

$$\rho_i = \frac{\eta_{max}^{(r)}}{\eta_{max}^{(i)}}, \quad \text{where } i \in [0, 2^l - 1], \tag{1}$$

and hence the damage factor ρ_r for random content is 1. Using the measurement results shown in Fig. 3 as an example, assume the BER_{max} is 5×10^{-3}, we calculate the four damage factors as $\rho_{11} = 0.33$, $\rho_{10} = 0.69$, $\rho_{00} = 1.01$, and $\rho_{01} = 1.58$.

3.2 Effect of Compression

We first derive the mathematical formulations for estimating the distribution characteristics of the compressed data and unused storage space size in each page. Let C_s denote the size of each uncompressed data sector (e.g., 4kB), m_s denote the number of uncompressed sectors in each page, and $C_p = m_s \cdot C_s$ denote the size of each flash memory page (e.g., 8kB). As pointed out above, the per-sector compression ratio x approximately follows a Gaussian distribution $N(\mu, \sigma^2)$. Let $m_s^{(e)}$ denote the number of compressed sectors per page when using explicit compression, and $C_s^{(e)}$ denote the length of the compressed data within one page. Due to the variation of the compression ratio x, both $m_s^{(e)}$ and $C_s^{(e)}$ are random variables. Since $x \cdot m_s^{(e)} \cdot C_s$ denotes the length of the compressed data within one page when $m_s^{(e)}$ is determined, $C_s^{(e)}$ can be expressed as

$$C_s^{(e)} = \sum_{m_s^{(e)}=m_s}^{\infty} x \cdot m_s^{(e)} \cdot C_s \cdot P\left(m_s^{(e)}\right), \tag{2}$$

where $P\left(m_s^{(e)}\right)$ is the probability that $m_s^{(e)}$ compressed sectors can fit into one page. We can express $P\left(m_s^{(e)}\right)$ as

$$P\left(m_s^{(e)}\right) = P\left\{x \cdot m_s^{(e)} \le m_s < x \cdot \left(m_s^{(e)} + 1\right)\right\} =$$
$$P\left\{x \cdot m_s^{(e)} \le m_s\right\} \cdot \left(1 - P\left\{x \cdot \left(m_s^{(e)} + 1\right) \le m_s\right\}\right).$$

Since $x \sim N(\mu, \sigma^2)$, we have that $x \cdot m_s^{(e)}$ and $x \cdot \left(m_s^{(e)} + 1\right)$ follow $N\left(\mu m_s^{(e)}, (\sigma m_s^{(e)})^2\right)$ and $N\left(\mu(m_s^{(e)} + 1), (\sigma(m_s^{(e)} + 1))^2\right)$, respectively. Hence, $P\left\{x \cdot m_s^{(e)} \le m_s\right\}$ and $P\left\{x \cdot \left(m_s^{(e)} + 1\right) \le m_s\right\}$ is the CDF (cumulative distribution function) for the random variant $x \cdot m_s^{(e)}$ and $x \cdot \left(m_s^{(e)} + 1\right)$. Accordingly, we have that

$$P\left(m_s^{(e)}\right) = \left[1 + erf\left(\frac{m_s - m_s^{(e)}\mu}{\sigma m_s^{(e)}\sqrt{2}}\right)\right] \cdot \left[1 - erf\left(\frac{m_s - (m_s^{(e)}+1)\mu}{\sigma(m_s^{(e)}+1)\sqrt{2}}\right)\right] / 4, \tag{3}$$

where $erf(z)$ is the error function for Gaussian distribution, i.e., $erf(z) = \frac{1}{\sqrt{\pi}} \int_{-z}^{z} e^{-t^2} dt$. For each given $m_s^{(e)}$, we can calculate the value of $P\left(m_s^{(e)}\right)$ based upon (3). Hence, each item in (2), i.e., $x \cdot m_s^{(e)} \cdot C_s \cdot P\left(m_s^{(e)}\right)$, is a random variable following a Gaussian distribution. As a result, we have that $C_s^{(e)} \sim N\left(\mu_{c_s^{(e)}}, \sigma^2_{c_s^{(e)}}\right)$, where

$$\begin{cases} \mu_{c_s^{(e)}} & = & \mu C_s \cdot \sum\limits_{m_s^{(e)}} m_s^{(e)} P\left(m_s^{(e)}\right), \\ \sigma^2_{c_s^{(e)}} & = & (\sigma C_s)^2 \cdot \sum\limits_{m_s^{(e)}} \left(m_s^{(e)} P\left(m_s^{(e)}\right)\right)^2. \end{cases} \tag{4}$$

When using implicit compression, the number of compressed sectors per flash memory page always remains as m_s and the length of compressed data per page is $C_s^{(i)} = x \cdot m_s \cdot C_s$. Therefore, we have that the random variable $C_s^{(i)} \sim N\left(\mu m_s C_s, \sigma^2 m_s^2 C_s^2\right)$.

3.3 Memory Damage Estimation

We further derive the mathematical formulations for calculating average memory cell damage per P/E cycle. Based upon the above discussions, we should consider four different design scenarios: (1) *UD*: unidirectional data layout without conditional data exchange, (2) *BD*: bidirectional data layout without conditional data exchange, (3) *UDC*: unidirectional data layout with conditional data exchange, (4) *BDC*: bidirectional data layout with conditional data exchange. Since the mathematical formulations can be derived with the same principle for all the scenarios, we first show the mathematical derivation in detail for *UD* (i.e., unidirectional data layout without conditional data exchange) and then present the results for the others without detailed derivations.

3.3.1 Derivation for the *UD* Design Scenario

We first define two parameters x_l and x_u as the ratios between the compressed data size and flash memory page size for lower and upper pages, respectively. Recall that both $C_s^{(e)}$ and $C_s^{(i)}$ (i.e., compressed data size within each flash memory page when using explicit and implicit compression, respectively) follow Gaussian distributions as derived in Section 3.2. Hence, x_l and x_u also follow the Gaussian distribution $N(\tilde{\mu}, \tilde{\sigma}^2)$, where $\tilde{\mu} = \mu_{c_s^{(e)}}/C_p$ and $\tilde{\sigma}^2 = \sigma^2_{c_s^{(e)}}/C_p^2$ for explicit compression, and $\tilde{\mu} = \mu m_s C_s/C_p$ and $\tilde{\sigma}^2 = \sigma^2 m_s^2 C_s^2/C_p^2$ for implicit compression. Define $z_l = \min(x_l, x_u)$ and $z_u = \max(x_l, x_u)$ and recall that $\{\rho_{11}, \rho_{10}, \rho_{00}, \rho_{01}\}$ represent the memory damage factors for the four different memory cell content and the damage factor ρ_r for random content is 1. In addition, let $m_s^{(c)}$ denote the average number of sectors per flash memory page and recall that m_s denote the number of sectors per flash memory page without using compression, and define $r = \frac{m_s^{(c)}}{m_s}$. Therefore, we can calculate the average memory cell damage per P/E cycle for the UD design scenario, which is normalized against the case of without using compression, as

$$\begin{aligned}
\rho_{UD} =& \frac{1}{r}\left(z_l + |x_l - x_u|\,\frac{\rho_{00}+2\rho_{11}+\rho_{10}}{4} + (1-z_u)\rho_{11}\right)\\
=& \frac{1}{r}\left(1 - \frac{\rho_{00}+2\rho_{11}+\rho_{10}}{4}\right) z_l\\
&+ \frac{1}{r}\left(\frac{\rho_{00}+\rho_{10}-2\rho_{11}}{4}\right) z_u + \frac{\rho_{11}}{r}\\
=& \frac{\lambda^l_{UD}}{r}\cdot z_l + \frac{\lambda^u_{UD}}{r}\cdot z_u + \frac{\rho_{11}}{r},
\end{aligned} \tag{5}$$

where

$$\lambda^l_{UD} = 1 - \frac{\rho_{00}+2\rho_{11}+\rho_{10}}{4},\ \lambda^u_{UD} = \frac{\rho_{00}+\rho_{10}-2\rho_{11}}{4}.$$

In order to obtain the distribution of ρ_{UD}, we must derive the distributions of z_u and z_l. The CDF of z_u can be written as

$$\begin{aligned}
F_{z_u}(z) &= P(x_l \le z, x_u \le z) = P(x_l \le z)\cdot P(x_u \le z)\\
&= F_{x_l}(z)\cdot F_{x_u}(z),
\end{aligned}$$

where F_{x_l} and F_{x_u} denote the CDF of x_u and x_l. Since x_u and x_l follow the same Gaussian distribution (denoted as f_N), we have that $F_{x_l} = F_{x_u}$. By taking the derivative of the CDF, we can obtain the PDF of z_u as

$$\begin{aligned}
f_{z_u}(z) &= F'_{z_u}(z) = f_N(z)\cdot\left(1 + erf\left(\frac{z-\tilde{\mu}}{\sqrt{2}\tilde{\sigma}}\right)\right)\\
&\approx f_N(z)\cdot\left(1 + \frac{z-\tilde{\mu}}{\sqrt{2}\tilde{\sigma}}\right).
\end{aligned} \tag{6}$$

Hence, f_{z_u} can be approximately expressed as the product of the PDF of a Gaussian distribution and a straight line with the slope of $\sqrt{2}\tilde{\sigma}$. Since $z \in (0,1]$, we could further approximate f_{z_u} to a PDF of a Gaussian distribution, i.e., $z_u \sim N\left(\mu_{z_u}, \sigma^2_{z_u}\right)$ and $f_{z_u}(z) = \frac{1}{\sigma_{z_u}\sqrt{2\pi}}\exp\left(-\frac{(z-\mu_{z_u})^2}{2\sigma^2_{z_u}}\right)$. The value of μ_{z_u} and σ_{z_u} can be obtained by solving

$$\begin{cases}
\left.\frac{d(f_{z_u}(z))}{dz}\right|_{z=\mu_{z_u}} \approx \left.\frac{d\left(f_N(z)\cdot\left(1+\frac{z-\tilde{\mu}}{\sqrt{2}\tilde{\sigma}}\right)\right)}{dz}\right|_{z=\mu_{z_u}} = 0,\\
f_{z_u}(z)|_{z=\mu_{z_u}} \approx f_N(z)\cdot\left.\left(1+\frac{z-\tilde{\mu}}{\sqrt{2}\tilde{\sigma}}\right)\right|_{z=\mu_{z_u}}.
\end{cases} \tag{7}$$

Accordingly, we have that

$$\begin{cases}
\mu_{z_u} = \tilde{\mu} + \frac{\sqrt{6}-\sqrt{2}}{2}\cdot\tilde{\sigma},\\
\sigma_{z_u} = \frac{2\sqrt{2}}{\sqrt{6}+\sqrt{2}}\cdot\tilde{\sigma}\cdot\exp\left(\frac{(\sqrt{6}-\sqrt{2})^2}{8}\right).
\end{cases} \tag{8}$$

We can obtain the PDF of z_l in similar manner. First, we can express the CDF of z_l as

$$\begin{aligned}
F_{z_l}(z) &= 1 - P(x_l > z, x_u > z)\\
&= 1 - (1 - P(x_l \le z))\cdot(1 - P(x_u \le z))\\
&= 1 - (1 - F_N(z))^2.
\end{aligned}$$

By taking the derivative of F_{z_l}, we obtain the PDF of z_l as

$$f_{z_l}(z) = 2(1 - F_N(z))\cdot f_N(z). \tag{9}$$

Similar to the above derivations for the case of z_u, we can approximate the PDF f_{z_l} as a Gaussian distribution $N\left(\mu_{z_l}, \sigma^2_{z_l}\right)$, where

$$\begin{cases}
\mu_{z_l} = \tilde{\mu} - \frac{\sqrt{6}-\sqrt{2}}{2}\cdot\tilde{\sigma},\\
\sigma_{z_l} = \frac{2\sqrt{2}}{\sqrt{6}+\sqrt{2}}\cdot\tilde{\sigma}\cdot\exp\left(\frac{(\sqrt{6}-\sqrt{2})^2}{8}\right).
\end{cases} \tag{10}$$

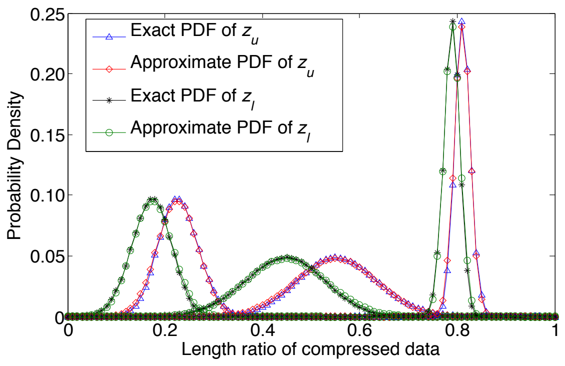


Figure 6: Comparison between the exact PDF of z_l, z_u and their Gaussian approximations with different sets of $\tilde{\mu}$ and $\tilde{\sigma}$.

To justify the Gaussian approximation of $f_{z_u}(z)$ and $f_{z_l}(z)$ (i.e., the PDF of z_u and z_l) in the above derivations, Fig. 6 compares the Gaussian approximation and the exact PDF, where we considered three different sets of $\{\tilde{\mu},\tilde{\sigma}\}$ (i.e., $\{0.2, 0.05\}$, $\{0.5, 0.1\}$, and $\{0.8, 0.02\}$, respectively) to cover a wide range of the compressed data length ratio and deviations. As clearly shown in Fig. 6, the Gaussian approximation of $f_{z_u}(z)$ and $f_{z_l}(z)$ incurs almost negligible inaccuracy.

Since $z_l \sim N\left(\mu_{z_l}, \sigma_{z_l}^2\right)$ and $z_u \sim N\left(\mu_{z_u}, \sigma_{z_u}^2\right)$, according to (5), the average cell damage ρ_{UD} also follows a Gaussian distribution, i.e., $\rho_{UD} \sim N\left(\mu_{UD}, \sigma_{UD}^2\right)$, where

$$\left\{ \begin{array}{lcl} \mu_{UD} & = & \frac{1}{r}\left(\lambda_{UD}^l \cdot \mu_{z_l} + \lambda_{UD}^u \cdot \mu_{z_u} + \rho_{11}\right), \\ \sigma_{UD} & = & \frac{1}{r}\sqrt{\left(\lambda_{UD}^l \cdot \sigma_{z_l}\right)^2 + \left(\lambda_{UD}^u \cdot \sigma_{z_u}\right)^2}. \end{array} \right. \quad (11)$$

Given the same data compressibility, the use of implicit and explicit compression leads to different distribution of x_u and x_l, and different r, leading to different memory cell damage.

3.3.2 More Formulation Results

Using the same principle, we can derive the mathematical formulations that can calculate the normalized average memory cell damage per P/E cycle for the other three design scenarios. Due to the page limit, we will directly present the final mathematical formulations without showing the derivation details.

For the *BD* design scenario that uses bidirectional data layout without conditional data exchange, its average memory cell damage is $\rho_{(BD)} \sim N\left(\mu_{(BD)}, \sigma_{(BD)}^2\right)$:

$$\left\{ \begin{array}{lcl} \mu_{(BD)} & = & \frac{1}{r}\left(\left(\lambda_{(BD)}^l + \lambda_{(BD)}^u\right) \cdot \tilde{\mu} + C_{(BD)}\right), \\ \sigma_{(BD)} & = & \frac{1}{r}\sqrt{(\lambda_{(BD)}^l)^2 + (\lambda_{(BD)}^u)^2} \cdot \tilde{\sigma}. \end{array} \right.$$

where

$$\left\{ \begin{array}{lcl} \lambda_{(BD)}^l & = & 1 - \frac{\rho_{11}+\rho_{10}}{2}, \quad \lambda_{(BD)}^u = 1 - \frac{\rho_{11}+\rho_{00}}{2}, \\ C_{(BD)} & = & \frac{2\rho_{11}+\rho_{10}+\rho_{00}}{2} - 1. \end{array} \right.$$

For the *UDC* design scenario that uses unidirectional data layout with conditional data exchange, its average memory cell damage is $\rho_{(UDC)} \sim N\left(\mu_{(UDC)}, \sigma_{(UDC)}^2\right)$:

$$\left\{ \begin{array}{lcl} \mu_{(UDC)} & = & \frac{1}{r}\left(\lambda_{(UDC)}^l \cdot \mu_{z_l} + \lambda_{(UDC)}^u \cdot \mu_{z_u} + \rho_{11}\right), \\ \sigma_{(UDC)} & = & \frac{1}{r}\sqrt{(\lambda_{(UDC)}^l \cdot \sigma_{z_l})^2 + (\lambda_{(UDC)}^u \cdot \sigma_{z_u})^2}. \end{array} \right.$$

where

$$\lambda_{(UDC)}^l = 1 - \frac{\rho_{11}+\rho_{10}}{2}, \quad \lambda_{(UDC)}^u = \frac{\rho_{11}+\rho_{10}}{2} - \rho_{11}.$$

For the *BDC* design scenario that uses bidirectional data layout with conditional data exchange, its average memory cell damage is $\rho_{(BDC)} \sim N\left(\mu_{(BDC)}, \sigma_{(BDC)}^2\right)$:

$$\left\{ \begin{array}{lcl} \mu_{(BDC)} & = & \frac{1}{r}\left(\lambda_{(BD)}^l \cdot \mu_{z_l} + \lambda_{(BD)}^u \cdot \mu_{z_u} + C_{(BD)}\right), \\ \sigma_{(BDC)} & = & \frac{1}{r}\sqrt{(\lambda_{(BD)}^l \cdot \sigma_{z_l})^2 + (\lambda_{(BD)}^u \cdot \sigma_{z_u})^2}. \end{array} \right.$$

3.4 Estimation of Memory Lifetime

This subsection discusses how we estimate the flash memory lifetime improvement based upon the average memory cell damage derived in the above section. In this work, we assume ideal wear-leveling, i.e., all the memory blocks always experience the same number of P/E cycles, and quantitatively define memory lifetime as the P/E cycle number that one memory block can survive before reaching the maximum allowable BER. Since it is common practice to use capacity over-provisioning in flash-based storage devices, we define an over-provisioning factor $\tau \geq 1$, i.e., the total physical storage capacity inside the storage device is $\tau\times$ larger than the storage capacity visible to the host. Let η denote the memory block P/E cycling endurance of the baseline scenario without using any data compression. Straightforwardly, the overall memory lifetime of the baseline scenario is $\tau \cdot \eta$ cycles.

Once data compression is used, the average memory cell damage becomes a random variable with a Gaussian distribution due to the Gaussian-like distribution of run-time data compression ratio. As a result, the cycling endurance of each memory block and hence overall memory lifetime also become random variables. Let $P_b^{(t)}$ denote the probability that one memory block can survive (i.e., can ensure the storage integrity even for incompressible data) after t P/E cycles, referred to as memory block survival probability. As the granularity of data erasure is in memory block units in NAND Flash memory,

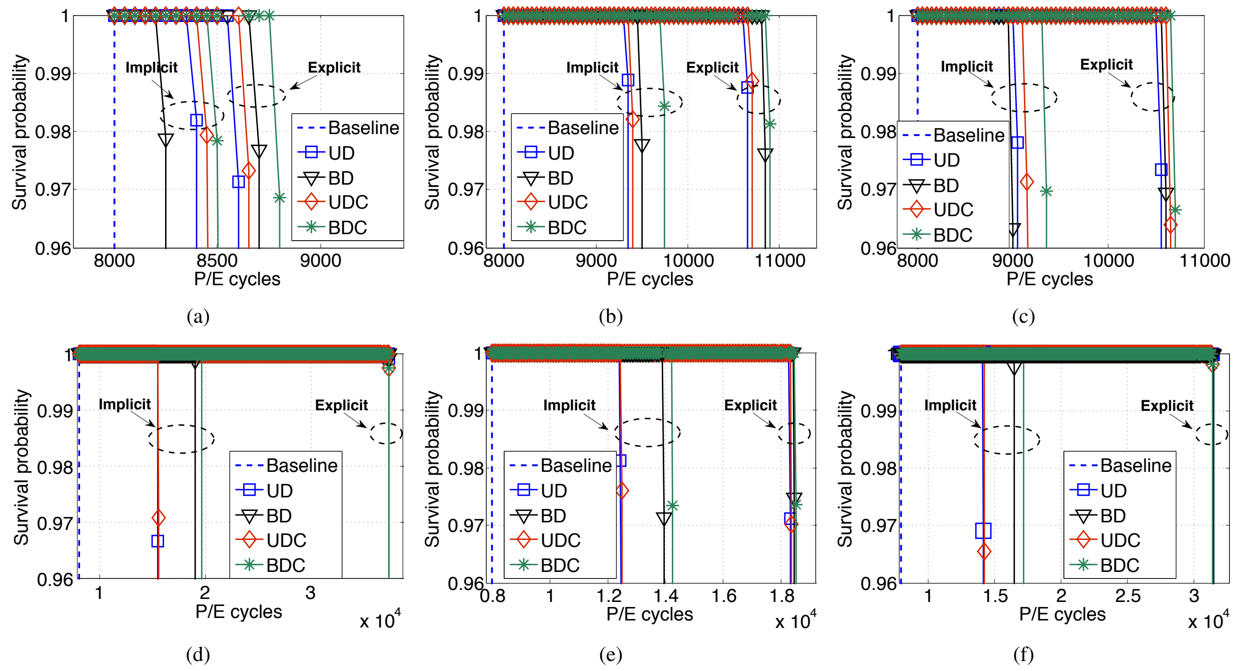


Figure 7: Storage device survival probability when storing (a) DLL, (b) Text, (c) Exe, (d) Log, (e) XML, and (f) HTML data in flash memory. Both explicit compression and implicit compression are considered.

the number of P/E cycles is independent among memory blocks. Hence, the survival of memory blocks is independent. Let N denote the number of memory blocks visible to the host, then the storage device contains $\tau \cdot N$ memory blocks in total. Therefore, once $P_b^{(t)}$ is known, based on the law of total probability, we can calculate the probability that the storage device can survive t cycles as

$$SP^{(t)} = \sum_{k=0}^{(\tau-1)N} \left(\binom{N\tau}{k} \cdot \left(P_b^{(t)}\right)^{N\tau-k} \cdot \left(1-P_b^{(t)}\right)^k \right), \tag{12}$$

which is called storage device survival probability. Suppose each memory block contains M wordlines and let $P_{wl}^{(t)}$ denote the survival probability of one wordline, we have that $P_b^{(t)} = \left(P_{wl}^{(t)}\right)^M$, i.e., one memory block survives only when all the wordlines inside this block survive. In the following, we will discuss how we can estimate the memory wordline survival probability $P_{wl}^{(t)}$.

For the baseline scenario without using data compression, the storage device fails to survive once the accumulated average damage of each memory cell reaches $\eta \cdot \rho_r$ (recall that $\rho_r = 1$ is the normalized memory cell damage factor when storing random data). When using data compression, let ρ_w denote the memory cell damage per cycle, where ρ_w could be $\rho_{(UD)}$, $\rho_{(BD)}$, $\rho_{(UDC)}$, or $\rho_{(BDC)}$ dependent upon the design strategies being used. By setting $\eta \cdot \rho_r$ as the maximum tolerable accumulated memory cell damage, we can express the P/E cycling endurance of each wordline as

$$T = \max(t), \;\; t \cdot \rho_w \leq \eta \cdot \rho_r - \rho_r. \tag{13}$$

Since ρ_w follows Gaussian distribution, $t \cdot \rho_w$ also follows Gaussian distribution with mean of $t \cdot \mu_{\rho_w}$ and variance of $t^2 \cdot \sigma_{\rho_w}^2$. Therefore, we can calculate the wordline survival probability $P_{wl}^{(t)}$ at t cycles as

$$P_{wl}^{(t)} = \frac{1}{2}\left(1 + erf\left(\frac{\tau \cdot \eta - 1 - t \cdot \mu_{\rho_w}}{t\sigma_{\rho_w}}\right)\right), \tag{14}$$

where μ_{ρ_w} and σ_{ρ_w} can be obtained using the formulations presented above for the four different design scenarios.

4 Quantitative Studies

With the formulations derived in Section 3, we studied the effectiveness of the design strategies presented in Section 2 for both explicit and implicit data compression. Based upon our measurement results with 20nm MLC NAND flash memory chips, we set the damage factors $\rho_{11} = 0.33$, $\rho_{10} = 0.69$, $\rho_{00} = 1.01$, and $\rho_{01} = 1.58$, as discussed in Section 3.1. As shown in (12), the storage device survival probability $SP^{(t)}$ depends on the over-provisioning factor τ and the total number of memory blocks N visible to the host. Assume the storage capacity of 512GB visible to the host and a block size of 4MB, we have N equals 128k. Each flash memory page has a size of 8kB, and we set the data sector size as 4kB. We further set the over-provisioning factor τ as 1.2. Based upon the memory chip measurement results and the over-provisioning factor of 1.2, we set the cycling endurance of the baseline scenario (i.e., without using data compression) as 8000.

4.1 Lifetime with Different Data Types

Using the measured compression ratio distribution of different data types as shown in Fig. 5, we evaluated the effectiveness of the developed design strategies on improving memory lifetime over the baseline scenario. Fig. 7 shows the results when using the four different design scenarios. Recall that the design scenario *UD* uses unidirectional data layout without conditional data exchange, *BD* uses bidirectional data layout without conditional data exchange, *UDC* uses unidirectional data layout with conditional data exchange, *BDC* uses bidirectional data layout with conditional data exchange. For the baseline scenario, the storage device lifetime remains 8000 regardless to the data types. When using data compression with different design strategies, the storage device lifetime becomes a random variable, whose CDF (i.e., its survival probability) is calculated according to the formulations derived in Section 3.

As shown in Fig. 7, explicit compression always outperforms implicit compression, which can be intuitively justified because explicit compression always tries to fit as many sectors as possible into each flash page. By comparing the data compressibility shown in Fig. 5 and the results shown in Fig. 7, we can clearly see that the difference between explicit compression and implicit compression strongly relies on the data compressibility. The higher data compressibility is, the larger difference between explicit compression and implicit compression is. In addition, the BDC design scenario always performs the best under both explicit and implicit data compression.

When explicit compression is being used, the difference among different design strategies tends to diminish for data with better compressibility (e.g., LOG and HTML). This can be explained as follows. With highly compressible data, explicit compression can fit more compressed data and hence leave less unused storage space within each flash memory page. As a result, there is a smaller room for these different design strategies to exploit the unused storage space, leading to almost the same storage device lifetime. On the other hand, when implicit compression is being used, unidirectional data layout and bidirectional data layout tend to have noticeable different effect, especially for data with better compressibility. As pointed out in Section 2.2.2, compared with bidirectional data layout, unidirectional data layout leads to more cells with random data content and '11'. Although '11' causes the least memory cell damage, random data tend to cause relatively large damage, as shown in Fig. 3. Based upon the content-dependent damage factors measured from our 20nm MLC flash memory chips, the penalty of having more random data can noticeably off-set the gain of having more '11'. As a result, unidirectional data layout tends to be inferior to bidirectional data layout. In the case of implicit compression, for data with worse compressibility (e.g., DLL and EXE), most memory cells would store random data content in both unidirectional and bidirectional data layout. As a result, bidirectional data layout will be inferior to unidirectional data layout in this scenario, which is shown in Fig. 7(a) and (c). Meanwhile, as shown in the results, the benefit of using conditional data exchange is not significant. Conditional data exchange aims to convert memory cell content from '00' to '10', since '10' causes less damage than '00'. Nevertheless, as shown in Fig. 3, the damage difference between '00' and '10' is not significant, which explains the low effectiveness of conditional data exchange observed in our study.

4.2 Sensitivity to Data Compressibility

The above results are based upon the measured compressibility characteristics of several different types of data. To more thoroughly elaborate on the impact of data compressibility, we carried out further evaluations by considering a much wider range of data compressibility in terms of compression ratio mean and standard deviation.

We first fix the data compression ratio standard deviation as 0.01, and Fig. 8 shows the corresponding storage device survival probability vs. lifetime for a wide range of compression ratio mean from 0.1 to 0.9. Data with better compressibility (i.e., smaller compression ratio mean) lead to larger lifetime improvement in both explicit compression and implicit compression. In addition, the advantage of explicit compression over implicit compression increases as the data have better compressibility. At the compression ratio mean of 0.1 (i.e., the data can be compressed by 10:1 on average), explicit and implicit compression can improve the storage device lifetime by 9.6 and 4.8 times, respectively. As shown in Fig. 8, for less compressible data (e.g., with the compression ratio mean of 0.7 and higher), explicit and implicit compression have almost the same effect. This is because, with low data compressibility, explicit compression can hardly increase the number of compressed data sectors per page. The results more clearly reveal the observations discussed above in Section 4.1: Under explicit compression, the difference between different de-

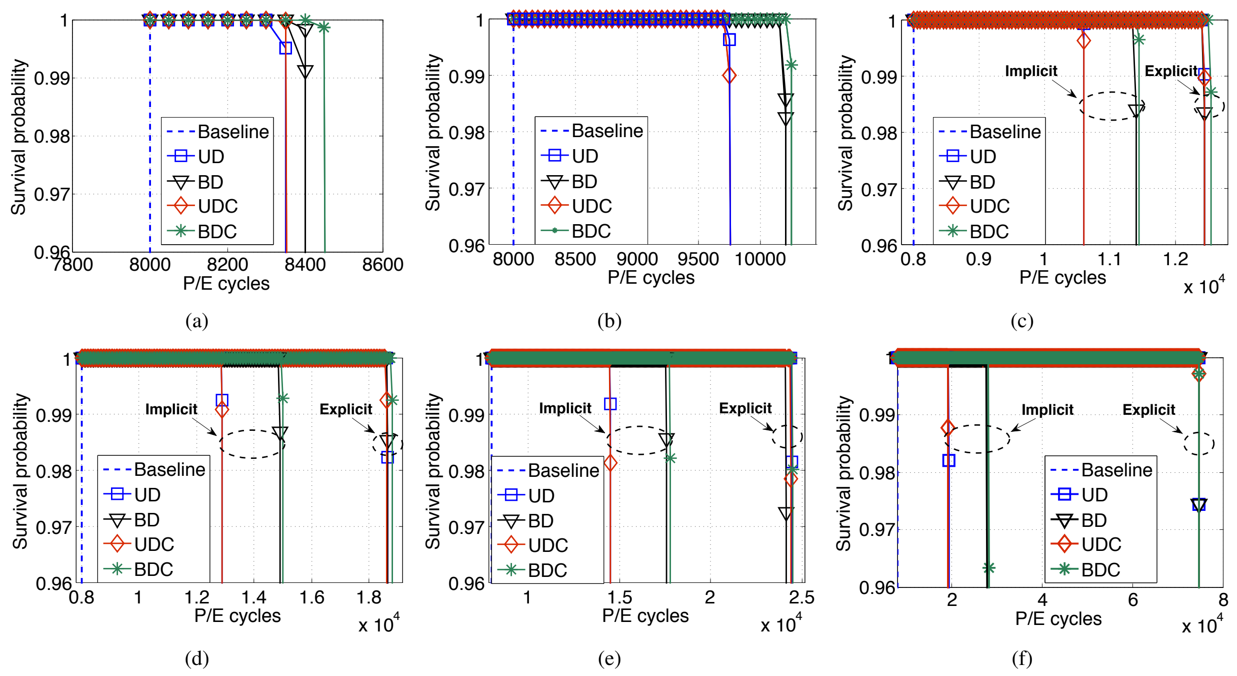


Figure 8: Storage device survival probability when data compression ratio standard deviation is 0.01 and mean is (a) 0.9, (b) 0.7, (c) 0.6, (d) 0.4, (e) 0.3, and (f) 0.1.

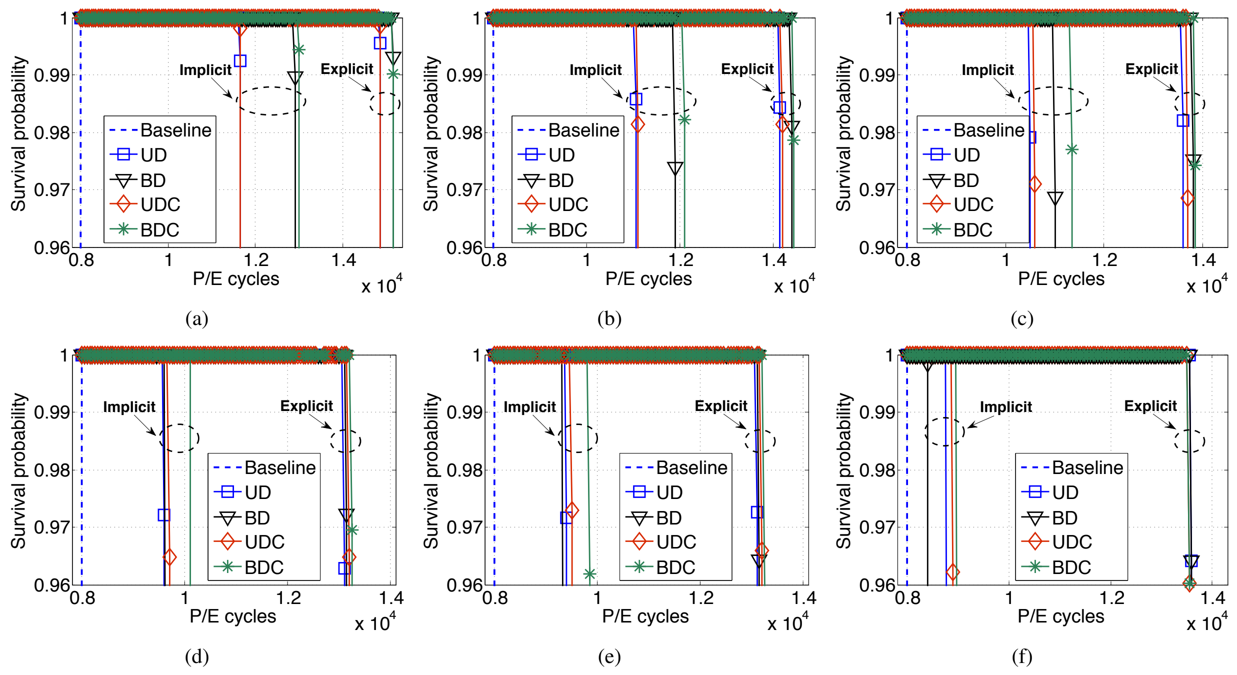


Figure 10: Storage device lifetime survival probability when compression ratio mean is 0.5 and standard deviation is (a) 0.01, (b) 0.03, (c) 0.05, (d) 0.09, (e) 0.1, and (f) 0.14.

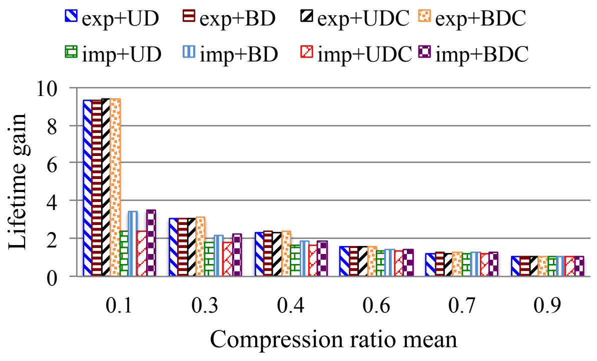


Figure 9: Lifetime gain under different data compression ratio mean.

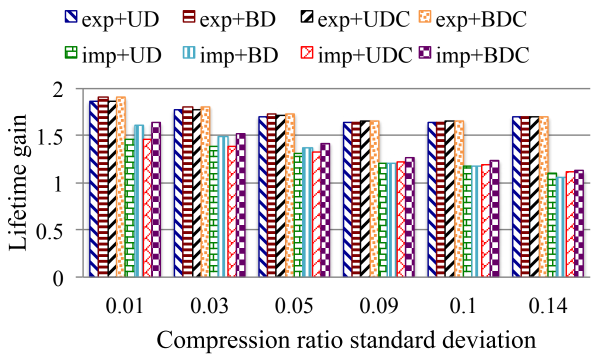


Figure 11: Lifetime gain for different data compression ratio standard deviation.

sign strategies quickly shrinks as we reduce the compression ratio mean; Under implicit compression, bidirectional data layout is always noticeably more beneficial than unidirectional data layout. By setting the storage device lifetime as the P/E cycles corresponding to 99.9% of storage device survival probability, Fig. 9 further plots the storage device lifetime gain over the baseline scenario without using compression under different compression ratio mean.

Next, we examined the impact of data compression ratio standard deviation. With the compression ratio mean of 0.5, Fig. 10 shows the storage device survival probability vs. P/E cycles when the compression ratio standard deviation varies from 0.01 to 0.14. As the data compression ratio standard deviation increases, advantage of explicit compression over implicit compression becomes more significant, and the storage device lifetime improvement generally reduces. In addition, the difference among the four different design scenarios reduces as the compression ratio standard deviation increases, for both explicit and implicit compression. Again, the design scenario of BDC is the most effective for both explicit and implicit compression.

By setting the storage device lifetime as the P/E cycles corresponding to 99.9% of storage device survival probability, Fig. 11 further shows the storage device lifetime gain over the baseline scenario under different compression ratio standard deviation. It shows that the lifetime gain monotonically reduces as we increase the data compression ratio standard deviation for implicit data compression. Nevertheless, for explicit data compression, the storage device lifetime gain first reduces and then saturates and even slightly increases as we increase the compression ratio standard deviation. The figure more clearly reveals the dependency of comparison between explicit and implicit compression on compression ratio standard deviation.

4.3 Discussions

The above quantitative studies show that the proposed implicit data compression is a viable complement to the conventional explicit data compression. Although explicit data compression may noticeably complicate the design of FTL and/or OS, it always outperforms implicit data compression from the storage device lifetime perspective. Nevertheless, the advantage of explicit compression over implicit compression strongly depends on the data compressibility. As shown in the above evaluation results, the advantage of explicit compression over implicit compression reduces as the data compressibility drops, and becomes very small as the data compression ratio mean becomes sufficiently large (e.g., over 0.6$\sim$0.7 in this study), particularly when the data compression ratio has a small standard deviation.

Our studies show that the bidirectional data layout outperforms the unidirectional data layout, especially when using the implicit data compression. Nevertheless, we should emphasize that this conclusion may not be always true. As pointed out above, compared with bidirectional data layout, unidirectional data layout result in more memory cells with random data content and '11'. Hence, which data layout option is better is fundamentally dependent on the exact values of the content-dependent damage factors. In this work, we extracted the content-dependent damage factors based upon measurements with 20nm MLC flash memory chips. However, for further scaled technology nodes such as 16nm or the emerging 3D flash memory, content-dependent damage factors and their relative comparison may (largely) change. This could essentially change the conclusion on the comparison between unidirectional data layout and bidirectional data layout. In addition, the above results suggest that the design strategy of conditional data exchange is not very effective, which is again also essentially due to the content-dependent damage factors being used in this work. For MLC NAND flash memory, the conditional data exchange will become more effective if the damage factors of '10' and '00' have a larger difference in future memory technology nodes.

Therefore, when applying the developed design framework in practice, one should carry out sufficient measurements and experiments to fully understand the content dependency of NAND flash memory damage and runtime data compressibility characteristics, in order to determine the most appropriate design strategy for leveraging data compressibility to improve device lifetime.

5 Related Work

Prior work [17–19] has studied the practical implementation of data compression in flash-based data storage systems, aiming to improve the storage system I/O speed performance and flash memory lifetime. In [17], a block-level compression engine is devised to support on-line compression for SSD-based cache, which is transparent to the file system. The authors of [18] develop a compression-aware FTL that can support compression-aware address mapping and garbage collection. The authors of [19] implement a caching system with commodity SSD by integrating data compression and data deduplication. All the prior work aimed to explicitly improve the storage efficiency, like the *explicit data compression* scenario being considered in this work. Besides data compression, prior work [20, 21] also investigated the practical implementation of data deduplication in flash-based storage systems.

FTL plays an important role in determining the lifetime of flash-based data storage devices, hence it has been well studied. The wear-leveling function in FTL aims to equalize the physical damage among all the flash memory block by appropriately allocating the memory blocks for erase and programming. A variety of techniques have been proposed to optimize the design of the wear-leveling function (e.g., see [10,22,23]). Aiming to reduce the write amplification and hence improve flash memory lifetime, the garbage collection function in FTL has been well studied (e.g., see [24]). The log-structured approach to managing flash memory have been considered through direct management of raw flash memory chips [11] or by facilitating the operation of the FTL inside SSDs [13]. Such log-structured file system level management of memory chips lead to improved flash memory lifetime and storage system performance.

The strength of fault tolerance, in particular ECC, also largely affect the storage device lifetime. Although classical BCH codes are still widely used in commercial flash-based storage devices [25, 26], the more powerful LDPC codes are receiving significant attention from the industry (e.g., see several industrial presentations at recent Flash Summit [2, 3, 27, 28]). A variety of techniques [29–32] have been developed to optimize the implementation of LDPC codes in future flash-based data storage devices.

6 Conclusion

This paper presents a thorough study on exploiting data compressibility to reduce cycling-induced flash memory cell physical damage and hence improve storage device lifetime. This work is essentially motivated by the content dependency of flash memory cell damage. We first present an unconventional implicit data compression strategy as a viable complement to explicit data compression being used in current practice, both of which represent different trade-offs between flash memory lifetime improvement and impact on FTL and system design complexity. In addition, their effectiveness and comparison largely vary with the runtime data compressibility characteristics. We further develop a set of design strategies that can exploit the unused storage space left by data compression within flash memory pages in order to minimize the overall memory physical damage. Furthermore, we derive a set of mathematical formulations that can quantitatively estimate the effectiveness of the proposed design strategies. Using 20nm MLC NAND flash memory chips, we carried out experiments to empirically evaluate the content dependency of flash memory cell damage. Employing these quantized experimental results, we compare the effectiveness of the proposed design strategies when using either explicit or implicit compression. Although this work focuses on flash memory, the proposed design strategies and developed mathematical formulations are readily applicable to other emerging memory technologies, e.g., PCM and ReRAM, that experience similar content dependency of memory cell damage.

Acknowledgements

We would like to thank our shepherd Sam H. Noh and the anonymous reviewers for their insight and suggestions for improvement. This work was supported by the National Science Foundation under Grants No. 1162152 and 1406154, National Science Foundation CAREER award CNS-125394, and the Department of Defense award W911NF-13-1-0157.

References

[1] G. Dong, N. Xie, and T. Zhang, "On the use of soft-decision error-correction codes in NAND Flash memory," IEEE Transactions on Circuits and

Systems I: Regular Papers, vol. 58, no. 2, pp. 429–439, 2011.

[2] E. Yeo, "An LDPC-enabled flash controller in 40nm CMOS," in Proceedings of Flash Memory Summit, 2012.

[3] X. Hu, "LDPC codes for Flash channel," in Proceedings of Flash Memory Summit, 2012.

[4] J. Kim, J. Kim, S. Noh, S. Min, and Y. Cho, "A space-efficient flash translation layer for compact flash systems," IEEE Transactions on Consumer Electronics, vol. 48, no. 2, pp. 366–375, 2002.

[5] K. Yim, H. Bahn, and K. Koh, "A flash compression layer for smart media card systems," IEEE Transactions on Consumer Electronics, vol. 50, no. 1, pp. 192–197, 2004.

[6] E. Gal and S. Toledo, "Algorithms and data structures for flash memories," ACM Computing Surveys, vol. 37, no. 2, pp. 138–163, 2005.

[7] J. Kang, H. Jo, J. Kim, and J. Lee, "A superblock-based flash translation layer for NAND Flash memory," in Proceedings of the 6th ACM & IEEE International conference on Embedded software, pp. 161–170, 2006.

[8] T. Park and J. Kim, "Compression support for flash translation layer," in Proceedings of the International Workshop on Software Support for Portable Storage, pp. 19–24, 2010.

[9] S. Lee, J. Park, K. Fleming, and J. Kim, "Improving performance and lifetime of solid-state drives using hardware-accelerated compression," IEEE Transactions on Consumer Electronics, vol. 57, no. 4, pp. 1732–1739, 2011.

[10] Y. Pan, G. Dong, and T. Zhang, "Error rate-based wear-leveling for NAND Flash memory at highly scaled technology nodes," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 21, no. 7, pp. 1350–1354, 2013.

[11] C. Manning, "Introducing yaffs, the first NAND-specific flash file system," http://linuxdevices.com, 2002.

[12] A. Schierl, G. Schellhorn, D. Haneberg, and W. Reif, "Abstract specification of the UBIFS file system for flash memory," in FM 2009: Formal Methods, pp. 190–206. Springer, 2009.

[13] C. Lee, D. Sim, J. Hwang, and S. Cho, "F2FS: A New File System for Flash Storage," in Proceedings of the 13th USENIX File and Storage Technologies (FAST), 2015.

[14] Y. Cai, E.F. Haratsch, O. Mutlu, and K. Mai, "Error patterns in MLC NAND flash memory: Measurement characterization and analysis," in Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 521–526, 2012.

[15] N. Mielke, T. Marquart, N. Wu, J. Kessenich, H. Belgal, S. Eric, F. Trivedi, E. Goodness, and L.R. Nevill, "Bit error rate in NAND Flash memories," in Proceedings of IEEE International Reliability Physics Symposium, pp. 9–19, 2008.

[16] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," IEEE Transactions on Information Theory, vol. IT-23, pp. 337–343, 1977.

[17] T. Makatos, Y. Klonatos, M. Marazakis, M. Flouris, and A. Bilas, "Using transparent compression to improve SSD-based i/o caches," in Proceedings of the European Conference on Computer Systems (EuroSys), pp. 1–14, 2010.

[18] S. Lee, J. Park, K. Fleming, Arvind, and J. Kim, "Improving performance and lifetime of solid-state drives using hardware-accelerated compression," IEEE Transactions on Consumer Electronics, vol. 57, no. 4, pp. 1732–1739, 2011.

[19] C. Li, P. Shilane, F. Douglis, H. Shim, S. Smaldone, and G. Wallace, "Nitro: A capacity-optimized SSD cache for primary storage," in Proceedings of USENIX Annual Technical Conference (ATC), pp. 501–512, 2014.

[20] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam, "Leveraging value locality in optimizing NAND Flash-based SSDs," in Proceedings of the 9th USENIX File and Storage Technologies (FAST), 2011.

[21] F. Chen, T. Luo, and X. Zhang, "CAFTL: A content-aware flash translation layer enhancing the lifespan of Flash memory based solid state drives," in Proceedings of the 9th USENIX File and Storage Technologies (FAST), 2011.

[22] Y. Lu, J. Shu, and W. Zheng, "Extending the lifetime of flash-based storage through reducing write amplification from file systems," in Proceedings of the 11th USENIX File and Storage Technologies (FAST), pp. 257–270, 2013.

[23] X. Jimenez, D. Novo, and P. Ienne, "Wear unleveling: improving NAND Flash lifetime by balancing page endurance," in Proceedings of the 12th USENIX File and Storage Technologies (FAST), 47–59, pp. 2014.

[24] L. Chang, T. Kuo, and S. Lo, "Real-time garbage collection for flash-memory storage systems of real-time embedded systems," ACM Transactions on Embedded Computing Systems, vol. 3, no. 4, pp. 837–863, 2004.

[25] Y. Lee, H. Yoo, I. Yoo, and I. Park, "6.4 gb/s multi-threaded BCH encoder and decoder for multi-channel SSD controllers," in Proceedings of IEEE International Solid-State Circuits Conference (ISSCC), pp. 426–428, 2012.

[26] H. Tsai, C. Yang, and H. Chang, "An efficient BCH decoder with 124-bit correctability for multi-channel SSD applications," in Proceedings of IEEE Asian Solid State Circuits Conference (A-SSCC), pp. 61–64, 2012.

[27] J. Yang, "The efficient LDPC DSP system for SSD," in Proceedings of Flash Memory Summit, 2013.

[28] L. Dolecek, "Non binary LDPC codes: The next frontier in ECC for flash," in Proceedings of Flash Memory Summit, 2014.

[29] J. Wang, T. Courtade, H. Shankar, and R. Wesel, "Soft information for LDPC decoding in flash: mutual-information optimized quantization," in Proceedings of IEEE Global Telecommunications Conference (GLOBECOM), 2011.

[30] S. Tanakamaru, Y. Yanagihara, and K. Takeuchi, "Over-10-extended-lifetime 76%-reduced-error solid-state drives (SSDs) with error-prediction LDPC architecture and error-recovery scheme," in Proceedings of IEEE International Solid-State Circuits Conference (ISSCC), 424–426, pp. 2012.

[31] J. Li, K. Zhao, J. Ma, and T. Zhang, "Realizing unequal error correction for NAND flash memory at minimal read latency overhead," IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 61, no. 5, pp. 354–358, 2014.

[32] J. Wang, K. Vakilinia, T. Chen, T. Courtade, G. Dong, T. Zhang, H. Shankar, and R. Wesel, "Enhanced precision through multiple reads for LDPC decoding in flash memories," IEEE Journal on Selected Areas in Communications, vol. 32, no. 5, pp. 880–891, 2014.

RAIDShield: Characterizing, Monitoring, and Proactively Protecting Against Disk Failures

Ao Ma[1], Fred Douglis[1], Guanlin Lu[1], Darren Sawyer[1], Surendar Chandra[2], Windsor Hsu[2]
[1]*EMC Corporation*, [2]*Datrium, Inc.*

Abstract

Modern storage systems orchestrate a group of disks to achieve their performance and reliability goals. Even though such systems are designed to withstand the failure of individual disks, failure of multiple disks poses a unique set of challenges. We empirically investigate disk failure data from a large number of production systems, specifically focusing on the impact of disk failures on RAID storage systems. Our data covers about one million SATA disks from 6 disk models for periods up to 5 years. We show how observed disk failures weaken the protection provided by RAID. The count of *reallocated sectors* correlates strongly with impending failures.

With these findings we designed RAIDSHIELD, which consists of two components. First, we have built and evaluated an active defense mechanism that monitors the health of each disk and replaces those that are predicted to fail imminently. This proactive protection has been incorporated into our product and is observed to eliminate 88% of triple disk errors, which are 80% of all RAID failures. Second, we have designed and simulated a method of using the joint failure probability to quantify and predict how likely a RAID group is to face multiple simultaneous disk failures, which can identify disks that collectively represent a risk of failure even when no individual disk is flagged in isolation. We find in simulation that RAID-level analysis can effectively identify most vulnerable RAID-6 systems, improving the coverage to 98% of triple errors.

1 Introduction

Storage systems have relied for decades on redundancy mechanisms such as RAID to tolerate disk failures, assuming an ideal world with independent and instantaneous failures as well as exponential distributions of the time to failure [3, 11, 18, 36]. However, some assumptions no longer hold given the fault model presented by modern disk drives. Schroeder and Gibson [42] analyzed 100,000 disks and rejected the hypothesis of the time between disk replacements following an exponential distribution. Further, in addition to whole-disk failures that make an entire disk unusable, modern drives can exhibit latent sector errors in which a block or set of blocks become inaccessible [6, 29]. Such sector faults in otherwise working disks further weaken the RAID reconstruction capability. Not only were sector errors previously ignored in the early RAID reliability model, these errors may worsen over time due to increasing drive complexity [4] and the common use of less reliable disks in storage systems [6, 17, 38]. In short, RAID protection is no longer enough; however, given its prevalence in the storage industry, a mechanism to shield RAID systems from unreliable disks would have a wide audience.

System designers have realized the new threats caused by these disk faults and built additional mechanisms to improve data reliability. While the original RAID system would protect against the loss of data from one disk (either an unavailable sector or the failure of the entire disk), the trend has been to use additional redundancy to guard against related data loss on multiple disks. For example, some storage arrays incorporate extra levels of parity, such as RAID-6, which can tolerate two simultaneous whole or partial disk failures [2, 12, 13, 19, 22, 23]; others add redundancy with CPU-intensive erasure coding [14, 25]. Throughout this paper we focus on "triple-disk failures," or "triple failures" for short, which refer to any combination of losing related data from three disks simultaneously, due to bad sectors or an entire disk. If a RAID-6 system encounters a triple failure it will lose data, but additional layers of redundancy (such as replication) can further protect against catastrophic data loss.

Many storage systems apply disk scrubbing to proactively detect latent sector errors; i.e., they read data from disk specifically to check for media errors, rather than because an application has requested the data [28, 43]. File systems also incorporate techniques such as replication and parity to improve data availability [10, 37, 41]; replication is critical because the failure of a disk group (DG) can be rectified, at high overhead, with a separate replica accessible via a LAN or WAN. Finally, even when primary storage systems are backed up onto separate dedicated backup systems, those backup systems can themselves be replicated [27].

Unfortunately, improvements to the basic RAID architecture are still based on certain assumptions given the limited understanding of disk fault modes. For example, empirical observations show both the sector error rate and the whole-disk failure rate grow over time [6, 42], causing RAID availability to continuously degrade. It is possible for multiple disks in the same RAID DG to fail simultaneously while other working disks have developed a number of latent sector errors [16]. Such multiple combined faults can overcome RAID protection and affect data availability. Unfortunately, little data is publicly available that quantifies such correlated faults.

To address this knowledge gap with respect to storage system reliability, we collected and analyzed disk error logs from EMC Data Domain backup systems. The data cover periods up to 60 months and include about 1 million SATA disks from deployed systems at customer and internal sites. To our knowledge, this is the first study of this magnitude to focus on analyzing disk faults (e.g., whole-disk failures and sector errors) that influence data reliability. The logs report when a disk failure is detected, at which point a system can automatically initiate data recovery onto a spare drive using available data from within that system. They also report larger-scale outages, when too many drives fail simultaneously for data to be accessible. We define a *recovery-related incident* as a failure that requires the retrieval of data from another system, such as a backup or disk replica.

Our analysis reveals that many disks fail at a similar age and the frequency of sector errors keeps increasing on working disks. Ensuring data reliability in the worst case requires adding considerable extra redundancy, making the traditional passive approach of RAID protection unattractive from a cost perspective. By studying numerous types of disk error, we also observe that the accumulation of sector errors contributes to whole-disk failures, causing disk reliability to deteriorate continuously. Specifically, a large number of *reallocated sectors* (RS[1]) indicates a high probability of imminent whole-disk failure or, at a minimum, a burst of sector errors.

With these findings we designed RAIDSHIELD, a monitoring mechanism, which proactively identifies and preempts impending failures and vulnerable RAID groups. RAIDSHIELD consists of two components, PLATE+ARMOR. First, we have built and evaluated *Predict Loss Accumulating in The Enterprise* (PLATE), an active defense mechanism that monitors the health of each disk by tracking the number of reallocated sectors, proactively detecting unstable disks and replacing them in advance. PLATE has been deployed in production systems for nearly a year. Second, we have designed and simulated *Assure Redundant Media Or Replace* (ARMOR), which uses the joint failure probability of a DG to quantify the likelihood of multiple simultaneous disk failures. ARMOR has the potential to identify sets of disks that collectively represent a risk of failure even when no individual disk is flagged in isolation. Given this assessment, unstable disks can then be replaced in advance or the redundancy of a DG can be increased; either approach can improve overall RAID availability.

[1]RS is also sometimes referred to as RAS in disk statistics, but we prefer to avoid the confusion with other uses of RAS in the CS literature.

Simulation results for PLATE, the single-disk proactive protection, show it can capture up to 65% of impending whole-disk failures with up to 2.5% false alarms. After incorporating it into our product, we find its effect on RAID failures is disproportionate: it has been observed to eliminate 70% of the recovery-related incidents caused by RAID failures and 88% of the RAID failures due to triple disk failures. Its benefits are somewhat limited by the types of errors that it cannot predict: about 20% of DG failures are caused by user errors, hardware faults, and other unknown reasons. Simulation results indicate that ARMOR, the cross-disk proactive protection, can effectively identify 80% of vulnerable RAID-6 systems in a test of 5500 DGs. We find that it can predict most of the triple failures not prevented by PLATE, leading to total coverage of 98% of triple failures.

The rest of this paper is organized as follows. We first provide background on partial disk failures and describe our storage system architecture, including an overview of RAIDSHIELD (§2). §3 presents our study on the relation between whole-disk failure and sector errors, and it characterizes reallocated sectors, which are found to be highly correlated with whole-disk failures. §4 describes and evaluates PLATE, demonstrating the substantial reduction in RAID failures after deploying single-disk predictive replacement. §5 describes the design and evaluation, via simulation, of ARMOR: using joint probabilities to assess the failure risk to a DG as a whole. §6 discusses related work and §7 concludes.

2 Background and Motivation

In this section we define disk partial failures, providing the background to understand our subsequent failure analysis. We then present an overview of our storage system architecture and describe the two aspects of RAIDSHIELD.

2.1 Disk Failures

Disks do not fail in a simple fail-stop fashion. Hence, there is no consensus definition of what constitutes a disk failure [5, 8, 45]. The production systems we studied define a whole-disk failure as:

- The system loses its connection to the disk,
- An operation exceeds the timeout threshold, or
- A write operation fails.

These criteria serve as the bottom line to replace disks that cannot function properly. However, in addition to whole-disk failures, disk drives can experience various partial failures while they still otherwise function. Sector-related issues are the major partial failures that endanger data safety [7, 31, 41]. Disk drives therefore provide a variety of proprietary and complicated mechanisms to rectify some failures and extend drive lifespans. In this subsection, we briefly describe disk technology, focusing on detection and error handling mechanisms for sector errors; refer elsewhere for more detailed descriptions [6, 38]. Failure detection and recovery mechanisms vary by manufacturer, production model, interface and capacity; the mechanisms introduced here cover common SATA disk internal mechanisms.

Sector errors can be categorized into different specific types based on how they are detected, as shown in Figure 1. Operations to the disk can be initiated by file system read() and write() calls as well as by an internal *scan* process, which systematically checks sector reliability and accessibility in the background. (These are shown in Figure 1 in blue, magenta, and green respectively.)

Media error: This error occurs when a particular disk sector cannot be read, whether during a normal read or a background disk scan. Any data previously stored in the sector is lost. The disk interface reports the status code upon detecting a sector error, specifying the reason why the read command failed.

Pending and Uncorrectable sector: Unstable sectors detected in the background process will be marked as pending sectors, and disk drives can try rectifying these errors through internal protection mechanisms, such as built-in Error Correcting Codes and Refreshment. These techniques rewrite the sector with the data read from that track to recover the faded data. Any sectors that are not successfully recovered will be marked as uncorrectable sectors.

Reallocated sector: After a number of unsuccessful retries, disk drives automatically re-map a failed write to a spare sector; its logical block address (LBA) remains unchanged. Modern disk drives usually reserve a few thousand spare sectors, which are not initially mapped to particular LBAs. Reallocation only occurs on detected write errors.

We also observe that changes to disk technology tend to increase the frequency of sector errors, a major fraction of partial disk failures. First, the number of sectors on a disk keeps increasing: while the capacity of individual disks may not be increasing at the rate once predicted by Kryder [33, 47], they still increase. Thus, if sector errors occur at the current rate, there would be more sector errors per disk. Second, the disk capacity increase comes from packing more sectors per track, rather than adding more physical platters. Sectors become increasingly vulnerable to media scratches and side-track erasures [15].

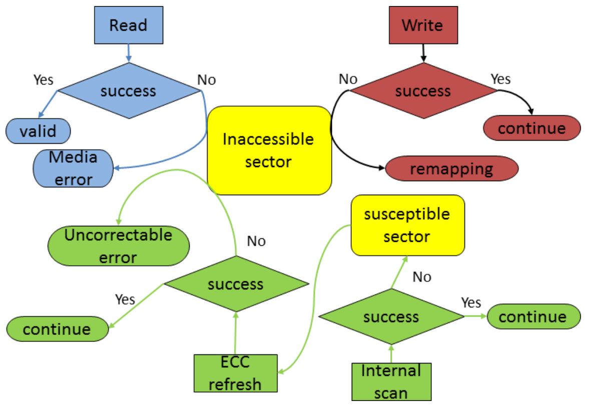


Figure 1: **Sector error transition.** *This figure depicts different responses to sector errors. A* read *(shown in blue) will report a media error if target sector is unreadable. A* write *(magenta) will attempt to remap a bad sector. An* internal scan *(green) will try to identify and rectify unstable sectors.*

2.2 Storage System Environment

We now briefly describe the context of our storage system with a focus on sector error detection and handling. At a high level, the storage system is composed of three layers, including a typical file system, the RAID layer, and the storage layer. The file system processes client requests by sending read and write operations to the RAID layer. The RAID layer transforms the file system requests into disk logical block requests and passes them to the storage layer, which accesses the physical disks. Our RAID layer adopts the RAID-6 algorithm, which can tolerate two simultaneous failures.

In addition to reporting latent sector errors captured in ordinary I/Os, our storage systems scrub all disks periodically as a proactive measure to detect latent sector errors and data corruption errors. Specifically, this scan process checks the accessibility of “live” sectors (those storing data accessible through the file system), verifies the checksums, and notifies the RAID layer on failures.

Sector error handling depends on the type of disk request. A failed write is re-directed to a spare sector through the automatic disk remapping process, without reporting the error to the storage layer. If a read fails, the RAID layer reconstructs data on the inaccessible sector and passes it to the storage layer for rewriting. Writing to the failed sector will trigger the disk internal mapping process. Note that given the process of RAID reconstruction and re-issued write, the failed sector detected through read (media error) will eventually lead to an RS. Therefore, the RS count is actually the number of inaccessible sectors detected in either reads or writes.

Finally, the systems evaluated in this paper are backup systems, which are known to have write-heavy workloads with fewer random I/Os than primary storage [46]; this workload may change the way in which disk faults are detected, as write errors may be relatively more common than read errors. The general conclusions should hold for other types of use.

2.3 RAIDSHIELD Motivation

Despite the expectation that RAID-6 systems should be resilient to disk failures, given a large enough population of DGs there will be errors leading to potential data loss [3]. Indeed, our systems encounter RAID-level errors, but thankfully these are extremely rare.[2] These systems usually rely on extra layers of redundancy such as (possibly off-site) replication to guard against catastrophic failures, but there is a strong incentive to decrease the rate at which RAID failures occur.

As we see in §3, disks that are installed together are somewhat likely to fail together, and disks that have partial (media) errors will rapidly accumulate errors until they are deemed to have failed completely. Our goal for RAIDSHIELD is to identify and replace failing disks before they completely fail, within reason. In the extreme case, one could use a single disk error as a warning signal and replace any disk as soon as it reported the slightest problem. However, the cost in time and expense would be prohibitive, especially for large-scale installations like cloud providers. With RAIDSHIELD, we take two tacks in this regard. The first is to use statistical information to discriminate between those disks that are likely to fail soon and those that are not. In the next section we consider a number of disk statistics that might be used for this purpose, finding that the *reallocated sectors* (RS) metric is an excellent predictor of impending failures. We show in §4 that after deploying PLATE proactive disk replacement, looking at each disk in isolation, our RAID failures dropped dramatically.

Can we do better with ARMOR, our second tack? We hypothesize that by using the joint failure probability across a DG we can find some additional instances where no single disk is close enough to failure to justify replacing it using the criteria for PLATE, but enough disks are symptomatic that the DG as a whole is in jeopardy. In §5 we present the probability analysis and some simulation results to justify this approach. In addition, we speculate that in some environments, it will be undesirable to proactively replace every disk that is showing the possibility of failure; instead, it may be important to prioritize among DGs and first replace disks in the most vulnerable groups. A single soon-to-fail disk in an otherwise healthy DG is a lower risk than a DG with many disks that have moderate probability of failure.

[2]We are unable to release specific error rates for DGs or disk models.

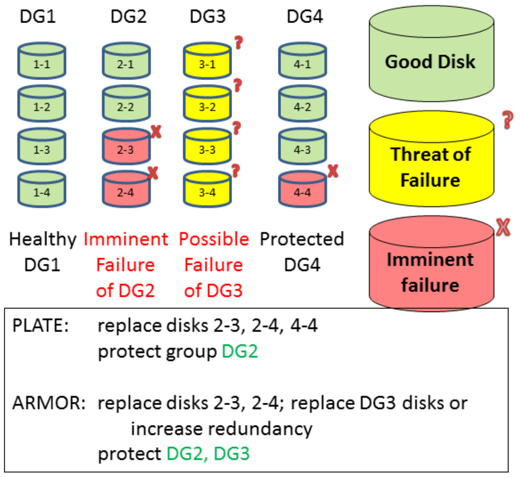


PLATE: replace disks 2-3, 2-4, 4-4
protect group DG2

ARMOR: replace disks 2-3, 2-4; replace DG3 disks or increase redundancy
protect DG2, DG3

Figure 2: **Example of RAIDSHIELD.** *Four DGs are shown, each with four disks. Green disks are healthy, yellow disks are at risk, and red disks are likely to fail imminently.* DG2 *and* DG3 *are at risk of failure.*

Figure 2 provides an example of the difference between PLATE and ARMOR. There are four disk groups; DG2, with two failing disks, is at high risk, while DG3 has a moderate risk due to the large number of partly-failing disks. With PLATE, we would replace the red disks, protecting vulnerable DG2 and improving the protection of DG4, but DG4 is already protected by three healthy disks. With ARMOR, we replace the two failing disks in DG2 but also recognize the vulnerability of DG3 given the large number of at-risk disks.

3 Disk Failure Analysis

Understanding the nature of whole-disk failures and partial failures is essential for improving storage system reliability and availability. This section presents the results of our analysis of about 1 million SATA disks. First, we describe how we collected the disk data studied in this work. Second, we present our observations of the new disk failure modes (e.g., simultaneous disk failures and sector errors) which endanger RAID availability. Third, we analyze the correlation between these two failure modes. Finally, we analyze characteristics and properties of reallocated sectors, the specific sector error type that is found to predict drive failures.

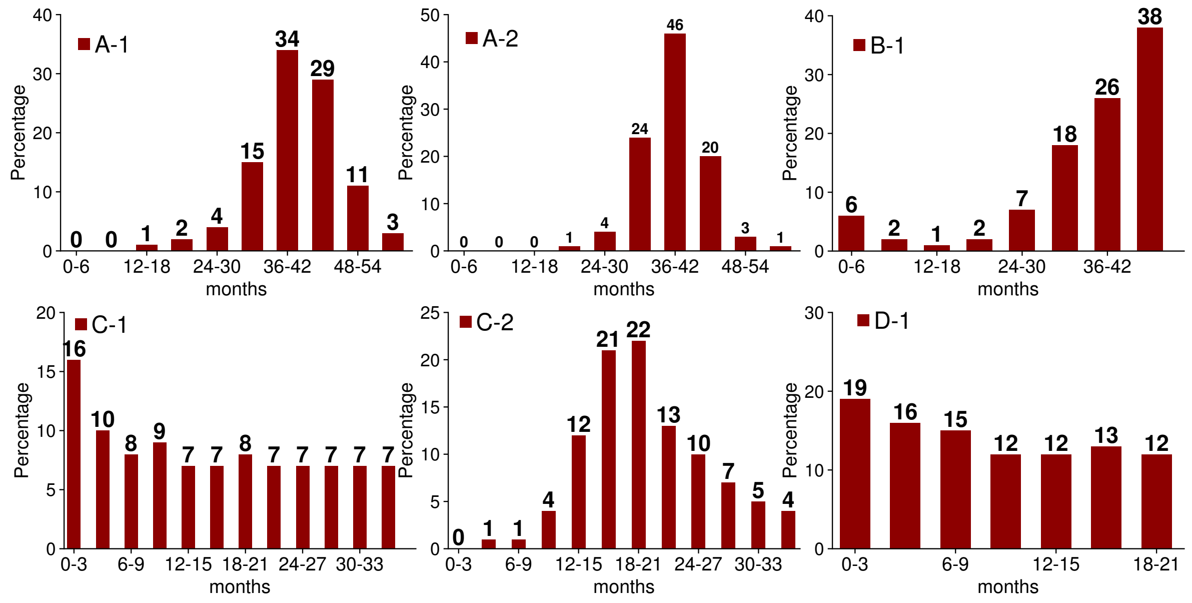


Figure 3: **Distribution of lifetimes of failed drives.** *These graphs show that many disks fail at a similar age. Note that the number of buckets, i.e. total age since deployment, and time length of each bucket varies by drive.*

Disk Model	Population (Thousands)	First Deployment	Log Length (Months)
A-1	34	06/2008	60
A-2	165	11/2008	60
B-1	100	06/2008	48
C-1	93	10/2010	36
C-2	253	12/2010	36
D-1	384	09/2011	21

Table 1: **Disk population.** *Population, earliest deployment date and log length of disk models used in this study.*

3.1 Data Collection

Our storage system has a built-in mechanism to log system status, which can optionally send important events back to a central repository each day [46]. These messages record a variety of system events including disk errors and failures. The data studied here are collected from these reports over a period of 5 years starting in June, 2008.

Similar to previous work [6], we anonymize disk information to make it possible to compare across disks from a single manufacturer but not across disk families. We denote each disk drive model as ⟨*family-capacity*⟩. Family is a single letter representing the disk family and capacity is a single number representing the disk's particular capacity. Although capacities are anonymized as a single number, relative sizes within a family are ordered by the number representing the capacity. That is, A-2 and C-2 are larger than A-1 and C-1 respectively.

Our entire sample of 1 million disks includes 6 disk models, each of which has a population of at least 30,000. They have been shipped in our storage systems since June, 2008, giving us a sufficient observation window to study various errors over the full lifespans of many drives. Details of the drives studied are presented in Table 1. Note that the recorded period of each disk model varies: the studied data range from 60-month logs of A-1 and A-2 down to 21 months for D-1.

3.2 New Disk Failure Modes

We observe two new disk failure modes that are not predicted by the early RAID reliability model and degrade RAID reliability and availability.

Drives fail at similar ages: We analyze all failed drives and categorize them into different buckets based on their lifetime. Figure 3 shows that a large fraction of failed drives are found at a similar age. For example, 63% of A-1 failed drives, 66% of A-2 failed drives and 64% of B-1 failed drives are found in their fourth year. This failure peak is also observed in the second year of the C-2 model, with 68% of failed drives found in this period. Given a large population of drives, some drives will fail not only in the same month but occasionally the same week or day, resulting in vulnerable systems. If a third error (a defective sector or a failed drive) should also occur before drives can be replaced and data reconstructed, the DG will be unavailable.

The lifetime distributions of C-1 and D-1 failed drives are comparatively uniform. However, these drives are

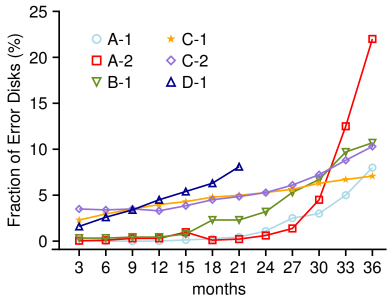


Figure 4: **Percentage of disks developing sector errors.** *As disks age, the number with at least one error increases, and the rate of increase is higher the older the disk is. Note that D-1 has only a 21-month record.*

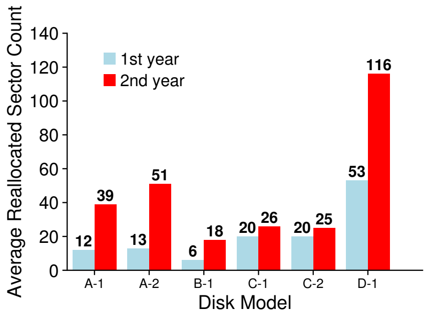


Figure 5: **Error counts year over year.** *Among disks with sector errors, for each model the number of errors increased significantly in the second year.*

relatively young compared to the drives with long observation intervals, so it is difficult to draw specific conclusions from this uniformity. We note a degree of "infant mortality" with these drives, with peaks of failures in the first three months.

Sector errors exacerbate risk: Figure 4 presents the fraction of disks affected by sector errors as a function of the disk age. Disks from all models show sector errors by the time they have been in use for 2–3 years, but some have significant errors much earlier. In addition, the rate at which errors appear increases with the age of the disks: for example, about 5% of A-2 disks get sector errors in the first 30 months, but it only takes an additional 6 months for 10% more to develop sector errors. Similar trends can be observed with A-1, B-1, and C-2.

To demonstrate the rate of error increase, we select 1000 disks randomly from each disk model, which developed at least one sector in a one-month observation window. We collect the count of their sector errors one year later. Figure 5 shows the average number of sector errors in the first and second years. For all drives with at least one sector error, the number of sector errors for the second year increases considerably, ranging from 25% for the C-2 model to about 300% for A-2.

These new disk failure modes reveal that the traditional RAID mechanism has become inadequate. The observation that many disks fail at a similar age means RAID systems face a higher risk of multiple whole-disk failures than anticipated. The increasing frequency of sector errors in working disks means RAID systems face a correspondingly higher risk of reconstruction failures: a disk that has not completely failed may be unable to provide specific sectors needed for the reconstruction. The disk technology trends introduced in §2.1 exacerbate these risks.

3.3 Correlating Full and Partial Errors

Since both whole-disk failures and sector errors affect data availability, exploring how they are correlated helps us to understand the challenges of RAID reliability. Here we introduce the statistical methodology used to analyze the data, then we evaluate the correlation between whole-disk failures and sector errors.

3.3.1 Statistical Methods

Our objective is to compare the sector errors in working disks and failed ones, and to use a measure to reflect their discrimination. We use quantile distributions to quantitatively evaluate the correlation degree between disk failures and sector errors. Specifically, we collect the number of sector errors on working and failed disks, summarizing each data set value using *deciles* of the cumulative distribution (i.e., we divide the sorted data set into ten equal-sized subsets; we normally display only the first nine deciles to avoid the skew of outliers). Such quantiles are more robust than other statistical techniques, such as mean and cumulative distribution function, to outliers and noise in depicting the value distribution and have been used to analyze performance crises in data centers [9].

3.3.2 Identifying Correlation

As introduced in §2.1, sector errors can be categorized into specific types based on how they are detected. For example, a sector error detected in a read is regarded as a media error while a sector error captured in a write is counted as an RS. Those error counts can be collected through the disk SMART interface [1] and are included in our logs.

Figures 6-7 compare the deciles of disk errors built on

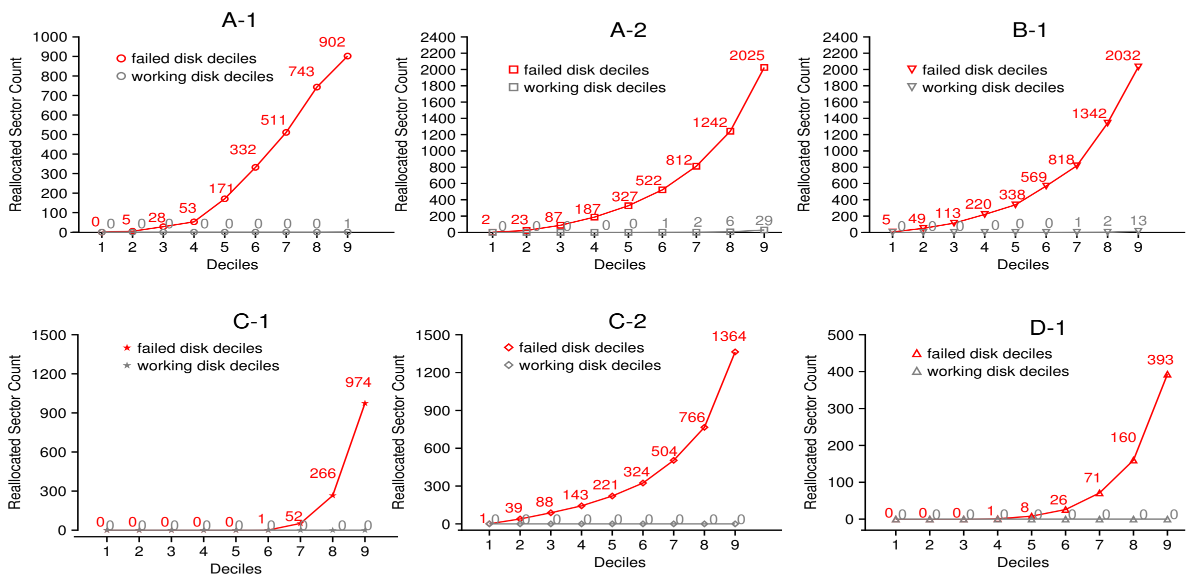


Figure 6: **Reallocated sector comparison.** *Failed drives have more RS across all disk models. Many disks fail before they exhaust their spare sectors. Failed drives with bigger capacity have more RS. Y-axis scales vary.*

the working and failed disk sets. The x-axis represents the Kth deciles, with the error counts on the y-axis.

Reallocated sector: Figure 6 presents the number of RS on failed and working drives. We observe that the majority of failed drives developed a large number of RS while most that are working have only a few. For example, 80% of A-2 failed drives have more than 23 RS but 90% of working drives have less than 29 of this error. Every disk model demonstrates a similar pattern; the only difference is how large the discrimination is. Failed disks have different RS counts, implying that many disks fail before they use up all spare sectors. We also find that failed drives with bigger capacity tend to have more RS, though the numbers depend more on the maximum number of reallocations permitted than the total size. For example, the median count of RS on A-2 failed drives is 327, compared to 171 for A-1; A-2 has both twice the capacity and twice the maximum number of reallocations, so this difference is expected. On the other hand, C-2 has twice the capacity as C-1 but the same maximum number of RS (2048), and its 9th decile of RS is only 40% higher than C-1. (Note that the median RS count for C-1 is zero, implying that many C-1 disks fail for reasons other than reallocated sectors; this is consistent with the large infant mortality shown in Figure 4 and bears further investigation. D-1 has similar characteristics.)

Media error: Due to the limitation of the logging messages we have on hand, we can analyze this error type only on the A-2 disk model. The result is presented in Figure 7. Though failed disks have more media errors than working ones, the discrimination is not that significant compared to RS. For example, 50% of failed disks have fewer than 15 media errors, and 50% of working ones developed more than 3 errors. There is a large overlap between them, perhaps because only sector errors detected in read operations are reported as media errors. Sector errors detected in writes will trigger the reallocation process directly without notifying the upper layer. Since the RAID layer will re-write the reconstructed data upon a detected media error, which causes the reallocation process, every media error will lead to an RS eventually: the media error count is thus a subset of RS. More details can be found in §2.2.

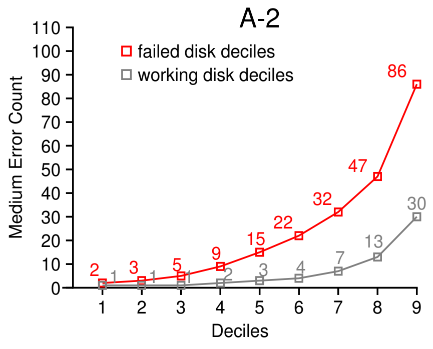


Figure 7: **Media error comparison.** *There is only moderate discrimination. Shown only for A-2.*

Pending and Uncorrectable sectors: As introduced in §2.1, sector errors discovered through the disk internal scan will be marked as pending sectors or uncorrectable sectors. The results for pending sectors are presented

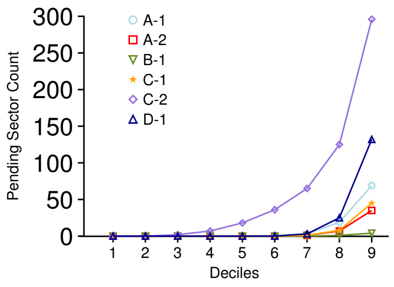


Figure 8: **Pending sector comparison.** *There is a large variation among different models.*

in Figure 8; the figure for uncorrectable sectors is similar and is omitted for space considerations. Through the comparison we find that for some disk models (such as A-1, A-2, and B-1), a certain fraction of failed disks (usually 30%) develop a similar amount of pending and uncorrectable sectors. Failed drives of the other disk models, including C-1, C-2, D-1 develop pending sector errors but none of them have uncorrectable sector errors, implying most pending errors have been addressed with drives' internal protection mechanisms. No working disks show these two types of sector errors, revealing that once disks develop these two types of error, they are very probable to fail.

3.3.3 Summary

These experiments characterize the correlation between whole-disk failures and various sector-related errors. We observe that most failed disks tend to have a larger number of RS than do working disks. Thus RS are strongly correlated with whole-disk failures. We infer that reallocation is the last resort to tolerate a defective sector after all other recovery mechanisms have failed; therefore, it avoids the influence of temporary errors which also appear on working disks. Further, given the process of RAID reconstruction and re-issued writes, inaccessible sectors detected through read and write will both eventually lead to RS. Therefore, the number of RS represents all inaccessible sectors and is a good indication of the extent to which a disk is wearing out.

3.4 Characterization of RS

The previous subsection revealed that RS appear more frequently in a large population of failed disks than working disks. Thus the number of RS is highly correlated with whole-disk failures across all disk models studied. This subsection studies characteristics of RS.

The best data set to study the properties of RS over the disk lifetime is disk model A-2. The reason is that this disk model was deployed for a long enough time period (more than 5 years) with a robust population of failed drives and detailed logging. Therefore, we use disk model A-2 as an illustration to explain our findings in the following sections.

All disks fail eventually, so we define an *impending disk failure* in our study as the disk failing within a 60-day observation window. A two-month window gives enough time for the disk to expose latent problems, since disk failure is not a simple fail-stop process. If a disk does not fail during this observation period, it is regarded as a qualified working disk.

We first evaluate how RS counts relate to disk failure rates. We analyze the percentage of disk failures after they exceed different thresholds of RS. The results are presented in Figure 9. The X-axis represents the RS count and the Y-axis depicts the failure percentage.

As found by Pinheiro, et al., the failure rate jumps dramatically once the disk starts to develop RS [38]. This rate grows steadily as the count of RS increases; for example, the failure rate of disks without any RS is merely 1.7%, while more than 50% of disks fail after this count exceeds 40. If the count grows to the range of 500 and 600, the failure rate increases to nearly 95%. We conclude that the more RS the disk has, the higher probability the disk will fail.

Second, we study the failed drives by analyzing the period between the time the disk RS count exceeds a certain value and the time a disk failure happens. We collect all time-to-fail (TTF) values and summarize the data set with the box-and-whisker plot in Figure 10, showing the 10-25-50-75-90 percentiles. All values for the time margin shrink as the number of RS grows. For example, one of every two failed disks would have more than seven days TTF when it exceeds 40 RS. But when the count of RS grows beyond 200, 50% of those disks that will soon fail are found to fail within just two days. However, the prediction is not guaranteed: the 90th percentile of failures is measured in weeks rather than days. We conclude that a larger number of RS indicates a disk will fail more quickly, in most cases just a few days.

Third, we analyze working drives, which have developed a certain number of RS, and categorize them into different buckets based on their RS counts. Figure 11 groups disks into buckets, randomly selecting 1000 disks with 0-100 RS, 1000 disks with 101-200 reallocations, and so on. We track how many sector errors they have accumulated 30 days later: for each bucket, the first (blue) bar shows the mean RS of the 1000 disks within that bucket as of the first month, and the second (magenta) bar shows the mean reallocations as of the second month.

The data shows that drives with less than 100 RS developed another 6 RS on average, while drives with RS in the range of 100 and 200 developed 100 more on average, well more than the aforementioned set. A similar

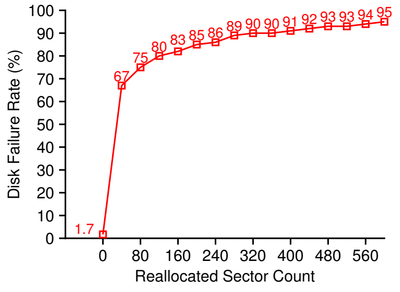


Figure 9: **Disk failure rate given different reallocated sector count.** *The failure probability increases quickly with more reallocated sectors. Shown for A-2.*

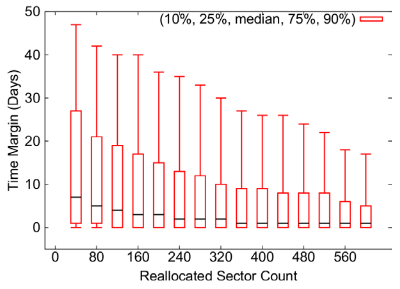


Figure 10: **Disk failure time given different reallocated sector count.** *The time margin decreases rapidly with more reallocated sectors. Shown for A-2.*

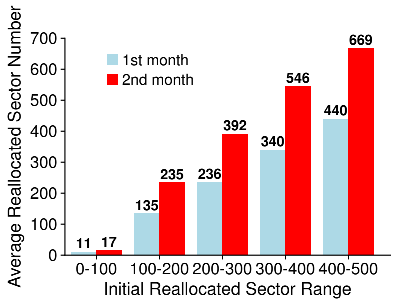


Figure 11: **Month-over-month comparison of reallocated sectors, grouped by first month's count.** *The mean reallocations in each bucket increase ˜50–75% across months. Shown for A-2.*

trend has been observed in other drive sets. In general, no matter how many existing sectors the disks have, the number of RS grows consistently.

Our analysis of other disk models is trending in the same direction of all the observations of model A-2. A slight difference is that the latest disk models can survive for a longer time with a certain number of RS. Therefore, the latest disk drives have a greater time margin as the number of RS grows.

From these experiments, we conclude that the accumulation of sector errors contributes to the whole-disk failure, causing disk reliability to deteriorate continuously. The more RS errors the drive has, the higher the probability to fail shortly or suffer a larger burst of sector errors. Therefore, the number of RS is a good criteria to reflect the disk survivability and sector reliability.

4 PLATE: Individual Failures

Much of the previous research on RAID has focused on improving redundancy schemes to tolerate more simultaneous failures [13, 30, 32, 39, 40]. However, our data analysis reveals that the likelihood of simultaneous whole-disk failures increases considerably with older disks. Further, the accumulation of sector errors contributes to whole-disk failures, causing the disk reliability to deteriorate continuously. Hence, ensuring data reliability in the worst case requires adding considerable extra redundancy, making the traditional passive approach of RAID protection unattractive from a cost perspective.

Meanwhile, the RS count has been observed to be a good criteria to quantify and predict the degree of deterioration of disk reliability. Therefore, we can upgrade the passive RAID protection into a proactive defense: PLATE monitors disk health (§4.1), identifies unreliable disks (§4.2), and replaces unstable disks in advance to prevent failures. Since unreliable disks are detected and removed promptly, the likelihood of simultaneous failures also decreases (§4.3).

4.1 Monitor Disk Status

Our previous analysis reveals that the number of RS is a good criteria to identify unstable disks. This expands the role of disk scrubbing: originally, scrubbing aimed to verify data accessibility and proactively detect lost data on failed sectors which could be recovered through RAID redundancy; thus, it only scans "live" sectors (those storing data accessible through the file system). The new findings show that recognizing *all* latent sector errors in a timely fashion is invaluable for monitoring the status of a DG, so our scrubbing is being updated to periodically check even unused disk sectors. We then monitor the status of each disk via daily system logs, and when a disk's RS count exceeds a threshold, its replacement is automatically triggered.

4.2 Proactively Identify Unreliable Disks

We see that the accumulation of sector errors contributes to whole-disk failures, causing disk reliability to deteriorate continuously. Hence, using the RS count can predict impending disk failures in advance. Such proactive protection provides administrators the chance to replace disks before whole-disk failures happen, improving RAID availability. We evaluate the methodology of the proactive protection through simulations based on historical disk information. We provide the result of deployment in production systems in §4.3.

If the RS count exceeds the given failure threshold T, the disk is considered to be unreliable. We evaluate the result using two curves that represent the trade-off between the fraction of failures successfully predicted

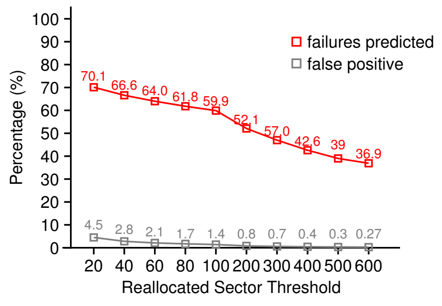


Figure 12: **Failure captured rate given different reallocated sector count.** *Both the predicted failure and false positive rates decrease as the threshold increases. Shown for A-2.*

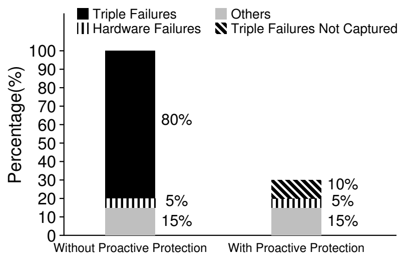


Figure 13: **Causes of recovery incidents.** *The distribution of causes of RAID failures, before and after proactive protection was deployed, normalized to the case without protection. Single disk proactive protection reduces about 70% of RAID failures and avoids 88% of the triple-disk failures previously encountered.*

(i.e., the *recall* of the prediction), and the false positive amount, which includes qualified working disks identified incorrectly. The impending whole-disk failure is defined as the disk failing within a 60-day observation window. If a disk that has more RS than the threshold is still functioning properly after the observation window, it is regarded as a false positive. Similarly, if a failed disk reports at least the given minimum number of RS within 60 days prior to the failure, the failure is successfully predicted. By comparing these two curves over the whole range of the identification threshold, we take into account all possible cost-based scenarios in terms of the trade-off between missing impending disk failures versus failing working ones incorrectly.

We measure the proactive protection on a population of 100,000 A-2 disks as reported by autosupport logs, and present the result in Figure 12. It shows that both the successful prediction rate and the false positive rate decrease smoothly as the RS threshold grows from 20 to 600. When the threshold is less than 200, it captures nearly 52–70% impending whole-disk failures, with 0.8–4.5% false positive rates. The majority of the unpredicted failures are caused by hardware faults, user error and other unknown reasons, which are unpredictable from a software perspective; these prediction rates are consistent with the curve for A-2 in Figure6, which depicted the fraction of failed disks that had encountered a given number of RS. Other disk models demonstrate similar trends in our experiments.

System administrators can decide the appropriate threshold to fail disks based on their expectation of captured rate, tolerance of replacing disks prematurely, and the time required to replace disks.

4.3 Deployment Result

PLATE, the single-disk proactive protection using remapped sector count, has been incorporated into some production systems. In our initial deployment, affecting disks A-1, A-2, and B-1, we set the threshold for predicting failure at 200 RS. This threshold was based on the "training set" of our analysis prior to deployment and was selected for two reasons: first, replacing disks in production systems may take up to 3 days in the worst case, and second, the median time to failure drops to less than 3 days when the count of RS grows beyond 200. In other words, setting the threshold less than 200 provides enough time to fix 50% of those impending failures proactively. In addition, the cost of replacing a working disk by mistake requires us to strive for a false positive rate less than 1% (i.e., $< 1\%$ unnecessarily added costs from incorrectly replacing working drives), resulting in a replacement threshold of at least 200.

Figure 13 compares the recovery incidents caused by RAID failures before and after proactive protection was added to our systems. The graphs are normalized to the average number of RAID failures per month before the deployment, which are dominated by triple failures (80%), the results of some combination of whole-disk failures and sector errors. Another 5% are due to other hardware faults (for example, errors in host bus adapters, cables and shelves), while the remaining 15% are caused by factors such as user error and other unknown reasons.

While it is a challenge to reduce failures due to hardware faults and other errors, single-disk proactive protection detects unstable drives before their reliability is further deteriorated and triggers the DG reconstruction promptly, reducing the likelihood of multiple simultaneous failures. We find this eliminates about 88% of recovery incidents caused by triple failures, equivalent to about 70% of all disk-related incidents. This disproportionate reduction in DG errors (relative to the fraction of individual disk failures we can predict) is because we only need to avoid one out of the three disk failures that would disable a RAID-6 DG.[3] The remaining 12% of triple failures are due to sudden failures or multiple "somewhat unreliable" disks, all of which have a number of RS but none of which exceeds the failure threshold; we address these in §5. All the proactively replaced disks subsequently undergo rigorous testing by our company upon their return; the specialists analyzing these disks have not seen a noticeable number of false positives upon replacement.

5 ARMOR: Multiple Failures

Single-disk proactive protection (PLATE) identifies and fails unreliable disks in advance, which can prevent potential data loss by reducing the likelihood of multiple simultaneous failures. But PLATE will wait patiently for one disk to exceed a threshold before sounding an alarm. Disks can fail quickly after exceeding that threshold and will sometimes fail before it is even reached. If several disks are close to being declared near imminent failure, they may collectively put the DG at high enough risk to take action. At the same time, simply replacing all unreliable disks is not the most efficient approach, because not every impending disk failure will lead to a RAID failure. If disk failures are within the tolerance of RAID redundancy, repair efforts may be better directed elsewhere: i.e., administrators might triage to prioritize another DG at higher risk. (Refer to the example in §2.3.)

The next subsection (§5.1) introduces how we quantify the degree of RAID reliability and identify a vulnerable RAID, which is likely to lose redundancy in the face of multiple unreliable disks. §5.2 presents some simulation results using the ARMOR technique, and §5.3 discusses ongoing work.

5.1 Identifying Vulnerabilities

The accumulation of sector errors contributes to whole-disk failures, causing the RAID reliability to deteriorate continuously. Therefore, we can quantify and predict the single disk reliability with its number of existing RS, which can be further used to evaluate the degree of RAID reliability deterioration through joint probability. There are two steps in this process.

Calculate the probability of single whole-disk failure: Our previous analysis reveals that the RS count reflects the likelihood of whole-disk failure. This probability is calculated as follows. We define:

- P(fail) as the probability of disk failure
- N_{RS} as the observed number of reallocated sectors
- P(N_{RS}) as the probability that a disk has a reallocated sector count larger than N_{RS}
- P(fail|N_{RS}) as the probability of a whole-disk failure given at least N_{RS} reallocated sectors
- P(N_{RS}|fail) as the probability that a failed disk has a reallocated sector count larger than N_{RS}

$$P(fail|N_{RS}) = \frac{P(N_{RS}|fail) \times P(fail)}{P(N_{RS})}$$
$$= \frac{\frac{num.\ of\ failed\ disks\ with\ N_{RS}}{num.\ of\ failed\ disks} \times \frac{num.\ of\ failed\ disks}{num.\ of\ disks}}{\frac{num.\ of\ all\ disks\ with\ N_{RS}}{num.\ of\ disks}}$$
$$= \frac{num.\ of\ failed\ disks\ with\ N_{RS}}{num.\ of\ all\ disks\ with\ N_{RS}}$$

Figure 14: **Formula of calculating the probability of whole-disk failure given a certain number of reallocated sectors.**

Ultimately we want to compute P(fail|N_{RS}), which can be calculated according to Bayes's Theorem (the first line of Figure 14).

Calculate the probability of a vulnerable RAID: Our storage system uses RAID-6, which can tolerate two simultaneous failures. We define *RAID vulnerability* as the probability of a RAID system having more than one disk failure. Specifically, we use the formula introduced in Figure 14 to calculate the failure probability of each disk given its reallocated sector count. The combination of these single disk probabilities allows us to compute RAID vulnerability using the formula shown in Figure 15. A similar methodology can be applied to other redundant disk systems (e.g., RAID-5).

5.2 Simulation Result

We evaluate our methodology of identifying vulnerable RAID DGs. Specifically, we analyze historical disk failures recorded in our logs and categorize their corresponding RAID DGs into two subsets: "good" RAID DGs with no disk failures (subset G) and "bad" RAID DGs with more than one disk failure (subset B). We use their reallocated sector counts (one or more days prior to a failure, in the case of subset B) as an input to compute the probability of RAID vulnerability through our measurement. If our approach can effectively identify vulnerable RAID DGs, the calculated probability of most DGs in subset B should be considerably larger than that of the majority of DGs in subset G.

We use one-year disk historical data to build the statistical model and collect 5000 DGs for G and 500 DGs for B respectively from other years. Deciles are used to summarize the distribution of vulnerable probability of these

[3] It may also arise from differences in the rate of failures over time, something that is difficult to assess.

$$P(vulnerable\ RAID|RS_1, RS_2, \ldots, RS_N) = P(\geq 2\ disks\ fail|RS_1, RS_2, \ldots, RS_N)$$
$$= 1 - P(0\ disk\ fail|RS_1, RS_2, \ldots, RS_N) - P(1\ disk\ fails|RS_1, RS_2, \ldots, RS_N)$$
$$P(0\ disk\ fail|RS_1, RS_2, \ldots, RS_N) = \prod_{i=1}^{N}(1 - P(i_{th}\ disk\ fails|RS_i))$$
$$P(1\ disk\ fails|RS_1, RS_2, \ldots, RS_N) = \sum_{j=1}^{N} P((j_{th}\ disk\ fails|RS_j) \prod_{i=1, i\neq j}^{N} (1 - P(i_{th}\ disk\ fails|RS_i))$$

N is the number of disks in a RAID DG, RS_i represents the reallocated sector count of disk i
$P(i_{th}\ disk\ fails|RS_i)$ represents the failure probability of i_{th} disk given RS_i reallocated sector count

Figure 15: **Formula of calculating the probability of a vulnerable RAID DG.**

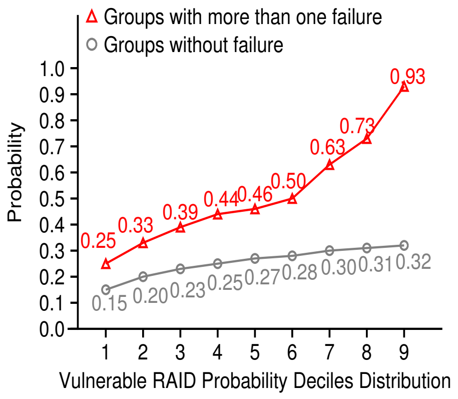


Figure 16: **Deciles Comparison of Vulnerable RAID probability.**

two subsets. The result is presented in Figure 16, which shows that probabilities of 90% of DGs in subset G are less than 0.32, while probabilities of most DGs in subset B are between 0.25 and 0.93. This probability discrimination between subset G and B show the methodology of identifying vulnerable RAID is effective to recognize endangered DGs, which are likely to have more than one disk failures. For example, when its probability grows to more than 0.32, we can regard this DG as a vulnerable one with high confidence. This threshold can capture more than 80% of vulnerable RAID DGs. Administrators can rely on this monitoring mechanism to keep track of disk statuses, recognize endangered RAID DGs, and trigger the appropriate proactive protection mechanism.

We examined the 12% of triple-disk failures that were not prevented by PLATE, looking at logs reported 3–7 days prior to the failure. (The duration varies depending on when the reports were made.) In 80% of the cases, ARMOR computed a failure probability of 80–95% despite no single disk being above the 200 RS threshold; this indicates that between PLATE and ARMOR, we could potentially prevent 98% of triple failures. Although the results of this analysis are based on a small sample, we are encouraged by the possibility of eliminating nearly failures resulting from triple-disk errors. However, greater attention will be needed for the 20% of RAID recovery incidents due to other causes.

5.3 Ongoing and Future Work

Incorporating the RAID monitoring mechanism into production systems has some operational considerations. We are upgrading our monitoring and logging mechanisms to recognize and record the reason for disk failure, as well as quantifying the variance of parameters of the statistical model, so we can activate the DG monitoring mechanism in our production systems.

We are considering methods to put a potentially failing disk "on probation" to test whether it is truly failing. This would be especially useful in cases where individual disks are not above a threshold for replacement but the availability of the DG as a whole is in doubt. Spare disks could be brought on-line while suspect disks get scrubbed thoroughly.

It would be interesting to extend ARMOR to other configurations, such as erasure coding, or to consider replicated data. A disk group or erasure coding system might be vulnerable when a given number of disks fail, but the data stored on it would be recoverable from another replica at a high cost. What if the replica is also vulnerable? The joint probability of multiple replicas failing simultaneously should by necessity be comparatively low, but it should be quantified.

Finally, it will be important to gain more operational experience with both PLATE and ARMOR on a greater variety of disk models over a greater period of time. Tuning the thresholds for the characteristics of each system will be important; even within a model, we would like to test different thresholds on a limited set of disks to determine the accuracy of our chosen threshold. Sensitivity to application workloads may also prove interesting: a backup storage system sees different read-write workloads than a primary system [46].

6 Related Work

Early work by Gibson, et al. [18, 36] and Chen, et al. [11] evaluates RAID reliability assuming an ideal world with independent failures, exponential lifetimes, and instantaneous failures. Unfortunately, the fault model presented by modern disk drives is more complex. Schroeder and Gibson [42] analyze 100,000 disks to reject the hypothesis that time between disk failure/replacement follows an exponential distribution. Bairavasundaram et al. [6] reveal the potential risk of sector errors during RAID reconstruction, which is not predicted in the early RAID reliability model. Researchers have since noted that the original RAID reliability model has outlived its useful life and built new models to depict RAID reliability [16, 21].

Given the presence of these new disk failure modes, many mechanisms have been built to improve system reliability. Schwarz, et al. [43] propose disk scrubbing to proactively detect latent sector errors. Many new storage arrays adopt extra levels of redundancy to tolerate more failures [12, 19]. File systems also detect and handle disk faults through checksums and replication. For example, in addition to using RAID techniques, ZFS employs checksums to detect block corruption and keep replicas of certain ”important” on-disk blocks to tolerate disk faults [10]. The IRON file system applies similar techniques to improve robustness of commodity file systems [41]. Another related approach is to tolerate disk faults at the application-level [44, 17].

Unfortunately, our previous analysis reveals that whole-disk failure and sector errors are strongly correlated. Further, the likelihood of such simultaneous failures is not consistent over time. Ensuring data reliability in the worst case requires adding considerable extra redundancy, which adds unnecessary costs.

Alternatively, a number of previous approaches seek indicators of impending failures. In particular, Pinheiro et al [38] study the failure characteristics of consumer-grade disk drives used in Google's services. They find that most SMART error metrics such as reallocated sectors strongly suggest an impending failure, but they also determine that half of failed disks show no such errors. We find that some disks (such as C-1 and D-1) frequently do not report errors before failing, but several models correlate well. These differences are presumably due to differences in disk models as well as workloads: since our systems rewrite data upon error, we may trigger remappings in ways their systems would not.

Goldszmidt [20] seeks to predict whole-disk failures with a performance signal, particularly the average maximum latency. Murray et al. [26, 34, 35] and Hamerly et al. [24] also attempt to improve whole-disk failure prediction by applying various advanced data mining algorithms on SMART [1] analytic data. In comparison, our work is on a much larger population of production disks with a focus on the correlation between whole-disk failures and sector errors, both of which affect the data safety. We quantitatively evaluate their correlation, and reveal that the RS count is a good criteria to reflect disk survivability and the sector reliability, which is then used to proactively recognize unstable disks and vulnerable RAID DGs.

7 Conclusion

In this paper, we present and analyze disk failure data from a large number of backup systems, including some of the world's largest enterprises. Our analysis reveals that the accumulation of *reallocated sectors*, a specific type of sector error, causes the disk reliability to deteriorate continuously. Therefore, the RS count can be used as an indicator to quantify and predict the degree of deterioration in disk reliability.

With these findings we designed RAIDSHIELD, consisting of PLATE and ARMOR. PLATE monitors disk health by tracking the number of RS and proactively detecting unstable disks; the deployment of single-disk proactive protection has eliminated 70% of RAID failures in production systems. With ARMOR, we aim to quantify the deterioration of RAID reliability and detect vulnerable RAID DGs in advance, even when individual disks have not degraded sufficiently to trigger alarms. Initial results with ARMOR suggest that it can eliminate most of the remaining triple-disk errors not identified by PLATE.

While we expect that the techniques presented here apply to all storage systems, the specific analyses were performed on backup systems with particular I/O patterns [46]. Extending the analysis and evaluating these techniques in other environments are promising future work.

Acknowledgments

We thank Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Stephen Manley, Philip Shilane, Grant Wallace, and members of the EMC Data Protection CTO office for comments and feedback on earlier drafts and discussions. We especially thank our shepherd, Haryadi Gunawi, and the anonymous reviewers; their feedback and guidance have been especially helpful.

In addition, we thank the Data Domain RAID, I/O, and customer support teams for feedback on the disk analyses and deployment. We thank Yanpei Chen, Rean Griffith, and Cheng Li for insightful comments on data science.

References

[1] B. Allen. Monitoring hard disks with S.M.A.R.T. Linux Journal, 2004.

[2] G. A. Alvarez, W. A. Burkhard, and F. Cristian. Tolerating multiple failures in RAID architectures with optimal storage and uniform declustering. In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA'97)*, pages 62–72, Denver, CO, USA, 1997.

[3] A. Amer, D. D. Long, and S. Thomas Schwarz. Reliability challenges for storing exabytes. In *Computing, Networking and Communications (ICNC), 2014 International Conference on*, pages 907–913. IEEE, 2014.

[4] D. Anderson, J. Dykes, and E. Riedel. More Than an Interface: SCSI vs. ATA. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST'03)*, pages 245–257, San Francisco, CA, USA, Apr. 2003.

[5] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. Fail-Stutter Fault Tolerance. In *The Eighth Workshop on Hot Topics in Operating Systems (HotOS VIII)*, pages 33–38, Schloss Elmau, Germany, May 2001.

[6] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. An analysis of latent sector errors in disk drives. In *Proceedings of the 2007 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '07)*, pages 289–300, San Diego, CA, USA, June 2007.

[7] L. N. Bairavasundaram, G. R. Goodson, B. Schroeder, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST '08)*, San Jose, California, February 2008.

[8] W. Bartlett and L. Spainhower. Commercial fault tolerance: A tale of two systems. *IEEE Trans. Dependable Secur. Comput.*, 1(1):87–96, Jan. 2004.

[9] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen. Fingerprinting the datacenter: Automated classification of performance crises. In *Proceedings of the 2010 EuroSys Conference (EuroSys '10)*, pages 111–124, Paris, France, Apr. 2010.

[10] J. Bonwick and B. Moore. Zfs: The last world in file systems. In *SNIA Software Developers's Conference*, Santa Clara, CA, Sept. 2008.

[11] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Comput. Surv.*, 26(2):145–185, June 1994.

[12] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-diagonal parity for double disk failure correction. In *Proceedings of the 3th USENIX Conference on File and Storage Technologies (FAST '04)*, page 14, San Francisco, CA, Apr. 2004.

[13] A. Dholakia, E. Eleftheriou, X.-Y. Hu, I. Iliadis, J. Menon, and K. Rao. A new intra-disk redundancy scheme for high-reliability raid storage systems in the presence of unrecoverable errors. *ACM Transactions on Storage*, 4(1):1:1–1:42, May 2008.

[14] C. Dubnicki et al. HYDRAstor: A scalable secondary storage. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST'09)*, 2009.

[15] J. Elerath. Hard-disk drives: the good, the bad, and the ugly. *Commun. ACM*, 52(6):38–45, June 2009.

[16] J. G. Elerath and J. Schindler. Beyond MTTDL: A closed-form RAID 6 reliability equation. *ACM Trans. Storage*, 10(2):7:1–7:21, Mar. 2014.

[17] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 29–43, Bolton Landing, NY, USA, Oct. 2003.

[18] G. Gibson. *Redundant disk arrays: Reliable, parallel secondary storage*. PhD thesis, University of California, Berkeley, CA, USA, 1992.

[19] A. Goel and P. Corbett. RAID triple parity. *ACM SIGOPS Oper. Syst. Rev.*, 46(3):41–49, Dec. 2012.

[20] M. Goldszmidt. Finding soon-to-fail disks in a haystack. In *USENIX HotStorage'12*, Boston, MA, USA, June 2012.

[21] K. M. Greenan, J. S. Plank, and J. J. Wylie. Mean time to meaningless: MTTDL, markov models, and storage system reliability. In *USENIX HotStorage'10*, Boston, MA, Oct. 2010.

[22] J. L. Hafner. Weaver codes: Highly fault tolerant erasure codes for storage systems. In *Proceedings*

of the 4th Conference on USENIX Conference on File and Storage Technologies (FAST'05), 2005.

[23] J. L. Hafner, V. Deenadhayalan, K. K. Rao, and J. A. Tomlin. Matrix methods for lost data reconstruction in erasure codes. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies (FAST'05)*, 2005.

[24] G. Hamerly and C. Elkan. Bayesian approaches to failure prediction for disk drives. In *ICML'01*, pages 202–209, Williamstown, MA, USA, June 2001.

[25] C. Huang et al. Erasure coding in windows azure storage. In *USENIX Annual Technical Conference*, 2012.

[26] G. F. Hughes, J. F. Murray, K. Kreutz-Delgado, and C. Elkan. Improved disk-drive failure warnings. *IEEE Transactions on Reliability*, 51(3):350–357, Sept. 2002.

[27] N. Jain, M. Dahlin, and R. Tewari. TAPER: tiered approach for eliminating redundancy in replica synchronization. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies (FAST'05)*, 2005.

[28] H. Kari, H. Saikkonen, and F. Lombardi. Detection of defective media in disks. In *IEEE Workshop on Defect and Fault Tolerance in VLSI Systems*, pages 49–55, Venice, Italy, Oct. 1993.

[29] H. H. Kari. *Latent Sector Faults and Reliability of Disk Arrays*. PhD thesis, Helsinki University of Technology, Espoo, Finland, May 1997.

[30] O. Khan, R. Burns, J. S. Plank, W. Pierce, and C. Huang. Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*, San Jose, February 2012.

[31] A. Krioukov, L. N. Bairavasundaram, G. R. Goodson, K. Srinivasan, R. Thelen, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Parity Lost and Parity Regained. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*, San Jose, California, February 2008.

[32] M. Li, J. Shu, and W. Zheng. Grid codes: Strip-based erasure codes with high fault tolerance for storage systems. *ACM Transactions on Storage*, 4(4):15:1–15:22, Feb. 2009.

[33] C. Mellor. Kryder's law craps out: Race to UBER-CHEAP STORAGE is OVER. *The A Register*, 2014. http://www.theregister.co.uk/2014/11/10/kryders_law_of_ever_cheaper_storage_disproven.

[34] J. F. Murray, G. F. Hughes, and K. Kreutz-Delgado. Hard drive failure prediction using non-parametric statistical methods. In *ICANN/ICONIP*, Istanbul, Turkey, June 2003.

[35] J. F. Murray, G. F. Hughes, and D. Schuurmans. Machine learning methods for predicting failures in hard drives: A multiple-instance application. In *Journal of Machine Learning research*, volume 6, pages 783–816, May 2005.

[36] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD Conference on Management of Data (SIGMOD'88)*, pages 109–116, 1988.

[37] H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman, and S. Owara. SnapMirror: file system based asynchronous mirroring for disaster recovery. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, 2002.

[38] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST '07)*, pages 17–28, San Jose, CA, USA, Feb. 2007.

[39] J. S. Plank and M. Blaum. Sector-disk (SD) erasure codes for mixed failure modes in RAID systems. *ACM Transactions on Storage*, 10(1), January 2014.

[40] J. S. Plank, M. Blaum, and J. L. Hafner. SD codes: Erasure codes designed for how storage systems really fail. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, San Jose, February 2013.

[41] 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'05)*, pages 206–220, Brighton, United Kingdom, Oct. 2005.

[42] B. Schroeder and A. G. Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST '07)*, San Jose, CA, Feb. 2007.

[43] T. J. Schwarz, Q. Xin, E. L. Miller, D. D. E. Long, A. Hospodor, and S. Ng. Disk scrubbing in large archival storage systems. In *IEEE MASCOTS'04*, pages 409–418, Volendam, The Netherlands, Oct. 2004. IEEE Computer Society.

[44] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[45] N. Talagala and D. Patterson. An Analysis of Error Behaviour in a Large Storage System. In *The IEEE Workshop on Fault Tolerance in Parallel and Distributed Systems*, San Juan, Puerto Rico, Apr. 1999.

[46] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu. Characteristics of Backup Workloads in Production Systems. In *Proceedings of the 10th Conference on File and Storage Technologies (FAST'12)*, 2012.

[47] C. Walter. Kryder's law. *Scientific American*, 293(2):32–33, 2005.

Write Once, Get 50% Free: Saving SSD Erase Costs Using WOM Codes

Gala Yadgar, Eitan Yaakobi, and Assaf Schuster
Computer Science Department, Technion
{gala,yaakobi,assaf}@cs.technion.ac.il

Abstract

NAND flash, used in modern SSDs, is a *write-once* medium, where each memory cell must be erased prior to writing. The lifetime of an SSD is limited by the number of erasures allowed on each cell. Thus, minimizing erasures is a key objective in SSD design.

A promising approach to eliminate erasures and extend SSD lifetime is to use *write-once memory (WOM)* codes, designed to accommodate additional writes on write-once media. However, these codes inflate the physically stored data by at least 29%, and require an extra read operation before each additional write. This reduces the available capacity and I/O performance of the storage device, so far preventing the adoption of these codes in SSD design.

We present *Reusable SSD*, in which invalid pages are reused for additional writes, without modifying the drive's exported storage capacity or page size. Only data written as a second write is inflated, and the required additional storage is provided by the SSD's inherent overprovisioning space. By prefetching invalid data and parallelizing second writes between planes, our design achieves latency equivalent to a regular write. We reduce the number of erasures by 33% in most cases, resulting in a 15% lifetime extension and an overall reduction of up to 35% in I/O response time, on a wide range of synthetic and production workloads and flash chip architectures.

1 Introduction

The use of flash based solid state drives (SSD) has increased in recent years, thanks to their short read and write latencies and increasing throughput. However, once flash cells are written upon, they must be erased before they can be rewritten. These comparatively slow erasures, along with the additional overheads they incur, significantly slow down pending read and write operations. In addition, flash cells have a limited *lifetime*, measured as the number of erasures a block can endure before its reliability deteriorates below an acceptable level [1, 12].

Erasures are the major contributors to cell wear [24]. Thus, much effort has been invested in attempts to reduce them and extend SSD lifetime. Suggested methods include minimizing write traffic [16, 18, 29, 38, 43, 46, 53] and distributing erase costs evenly across the drive's

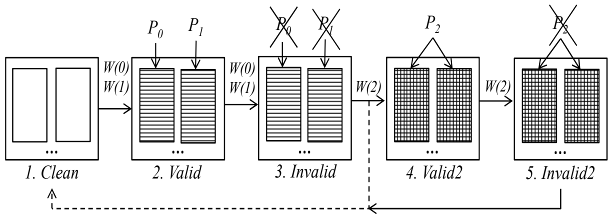


Figure 1: A simplified depiction of our second write approach. Each block holds one page, and begins in a clean state (1). Logical pages P_0 and P_1 are written by the application and stored on the first two blocks of the drive (2). When they are written again their copies are invalidated (3) and written elsewhere (not shown). Normally, the blocks would now be erased and returned to the clean state. Instead, in our design, they are *reused* to write logical page P_2 as a *second write* (4). When page P_2 is written again by the application, its copy is invalidated (5) and the blocks are erased, returning to the clean state.

blocks [1, 20, 27, 28]. While most of these methods improve performance due to the reduction in erasures, others extend device lifetime at the cost of degrading performance [11, 24, 30]. Another approach is to improve current error correction methods in order to compensate for the decreasing reliability of blocks late in their lifetime [7, 44, 60].

A promising technique for reducing block erasures is to use write-once memory (WOM) codes. WOM codes alter the logical data before it is physically written, thus allowing the reuse of cells for multiple writes. They ensure that, on every consecutive write, zeroes may be overwritten with ones, but not vice versa. WOM codes were originally proposed for write-once storage media such as punch cards and optical disks [47]. However, they can be applied to flash memories, which impose similar constraints: the bit value of each cell can only increase, not decrease, unless the entire block is erased[1]. Indeed, several recent studies proposed the use of WOM codes to reduce SSD block erasures [4, 14, 21, 23, 35, 42, 57].

Unfortunately, the additional writes come at a price. The old data must be read before the new data can be encoded. More importantly, WOM codes 'inflate' the data: the physical capacity required for storing the encoded data is larger than the original, logical, data by at least 29% —

[1] We adopt the conventions of coding literature, and refer to the initial, low voltage state of flash cells as zero.

a theoretical lower bound [17, 47]. Furthermore, WOM code design involves three conflicting objectives: minimizing physical capacity, minimizing encoding complexity, and minimizing the probability of encoding failure. Any two objectives can be optimized at the cost of compromising the third.

Existing studies have focused on minimizing the physical capacity overhead and the probability of encoding failure, greatly increasing complexity. The resulting designs, in which additional writes are performed on the same 'used' page, incur impractically high overheads. Thus, the industry has been unable to exploit recent theoretical advances effectively.

Our goal is to bridge the gap between theory and practice. To achieve a practical, applicable design, we are willing to tolerate a certain probability of (albeit rare) encoding failure and mitigate its penalties with the negligible overhead of an additional calculation.

We present *Reusable SSD* — a design that uses WOM codes to perform second writes on flash, thus reducing erasures and extending SSD lifetime. This is, to the best of our knowledge, the first design that addresses all the practical constraints of WOM codes. A simplified depiction of the design of Reusable SSD appears in Figure 1.

In order to preserve the SSD's logical capacity, we perform first writes with unmodified logical data, with no additional overheads, and utilize existing spare space within the SSD to perform 'inflated' second writes. For efficient storage utilization, we use second writes only for "hot" data that is invalidated quickly.

In order to preserve the SSD's I/O performance, we use WOM codes with encoding complexity equivalent to that of error correction codes used by current SSDs. Second writes are written on two used physical pages on different blocks, so that they are read and written in parallel, avoiding additional latency. We prefetch the old, invalid, data in advance to avoid additional delays.

We evaluate Reusable SSD using the SSD [1] extension of the DiskSim simulator [5], and a well-studied set of workloads [40, 41]. Second writes in our design are indeed shown to be "free": they reduce the number of erasures by an almost steady 33%, resulting in a 15% lifetime extension. By eliminating so many erasures while preserving the read and write latency of individual operations, our design also notably reduces I/O response time: up to 15% in enterprise architectures and up to 35% in consumer class SSD architectures. Furthermore, our design is orthogonal to most existing techniques for extending SSD lifetime. These techniques can be combined with Reusable SSD to provide additional lifetime extension.

The rest of this paper is organized as follows. Section 2 contains the preliminaries for our design. We present our implementation of second writes in Section 3, and give an overview of our design in Section 4. The details are described in Section 5, with our experimental setup and evaluation in Section 6. We survey related work in Section 7, and conclude in Section 8.

2 Preliminaries

2.1 Use of NAND Flash for SSD

A flash memory chip is built from floating-gate cells that can be *programmed* to store a single bit, two bits, and three bits in SLC, MLC and TLC flash, respectively. Cells are organized in blocks, which are the unit of erasure. Blocks are further divided into pages, which are the read and program units. Each block typically contains 64-384 pages, ranging in size from 2KB to 16KB [12, 14]. Within the chip, blocks are divided into two or more *planes*, which are managed and accessed independently. Planes within a chip can operate concurrently, performing independent operations such as read, program, and erase, possibly with some minor restrictions [14, 19, 49, 55].

Each page is augmented with a *page spare area*, used mainly for storing redundancy bytes of *error correction codes* (ECC) [12, 14]. The size of the spare area ranges between 5% and 12.5% of the page's logical size [19, 49, 55, 60]. The larger sizes are more common in recent architectures, because scaling in technology degrades the cell's reliability [12, 45]. Furthermore, the *bit error rate (BER)* increases as a function of the block's lifetime, requiring stronger ECC as the block grows older [8, 12, 37].

Write requests cannot update the data in the same place it is stored, because the pages must first be erased. Thus, writes are performed out-of-place: the previous data location is marked as invalid, and the data is written again on a clean page. To accommodate out-of-place writes, some physical storage capacity is not included in the drive's exported logical capacity. Thus, the drive's *overprovisioning* is defined as $\frac{T-U}{U}$, where T and U represent the number of physical and logical blocks, respectively [12]. Typical values of overprovisioning are 7% and 28% for consumer and enterprise class SSDs, respectively [52]. The *Flash Translation Layer (FTL)* is responsible for mapping logical addresses to physical pages.

Whenever the number of clean blocks drops below a certain threshold, the *garbage collection* process is invoked. Garbage collection is typically performed greedily, picking the block with the minimum *valid count* – number of valid pages, as the victim for *cleaning*. The valid pages are *moved* – read and copied to another available block, and then the block is erased. The addition of internal writes incurred by garbage collection is referred to as *write amplification* [12]. It delays the cleaning process, and requires, eventually, additional erasures. Write amplification can be reduced by increasing overprovisioning, sacrificing logical capacity for performance and block lifetime [12, 43, 52].

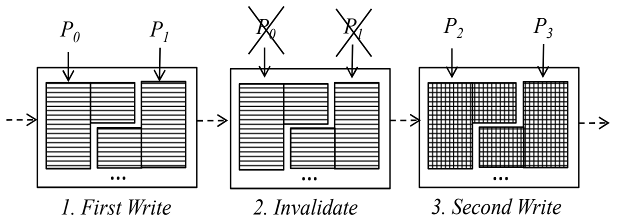


Figure 2: Naive implementation of second writes with the code in Table 1. Every write, first or second, must program and occupy storage capacity equivalent to 150% of the logical page size.

Data on the drive is usually not updated uniformly. Thus, some blocks may reach their lifetime limit, rendering the drive inoperable, while many blocks are still 'young'. Several techniques have been proposed for *wear leveling* – distributing erasures uniformly across the drive's blocks [1, 27].

2.2 Write-Once Memory Codes

Write-once memory (WOM) codes were first introduced in 1982 by Rivest and Shamir, for recording information multiple times on a write-once storage medium [47]. They give a simple WOM code example, presented in Table 1. This code enables the recording of two bits of information in three cells twice, ensuring that in both writes the cells change their value only from 0 to 1. For example, if the first message to be stored is 11, then 001 is written, programming only the last cell. If the second message is 01, then 101 is written, programming the first cell as well. Note that without special encoding, 11 cannot be overwritten by 01 without prior erasure. If the first and second messages are identical, then the cells do not change their value between the first and second writes. Thus, before performing a second write, the cell values must be *read* in order to determine the correct encoding.

Data bits	1st write	2nd write
00	000	111
10	100	011
01	010	101
11	001	110

Table 1: WOM Code Example

In the example from Table 1, a total of four bits of information are written into three cells: two in each write. Note that on a write-once medium, a basic scheme, without encoding, would require a total of four cells, one for every data bit written. In general, the main goal in the design of WOM codes is to maximize the total number of bits that can be written to memory for a given number of cells and number of writes. The number of bits written in each write does not have to be the same.

A WOM code design is a called a *construction*. It specifies, for a given number of cells, the number of achievable writes, the amount of information that can be written in each write, and how each successive write is encoded. Numerous methods have been suggested for improving WOM code constructions [4, 6, 21, 25, 47, 51, 56].

To see how WOM codes can be used to reduce erasures, consider a naive application of the WOM code in Table 1 to SSD. Every page of data would be encoded by the SSD controller into 1.5 physical pages according to the WOM code construction from Table 1. Thus, each page could be written by a first write, invalidated, and written by a second write before being erased, as depicted in Figure 2. Such an application has two major drawbacks: (1) Although additional writes can be performed before erasure, at any given moment the SSD can utilize only 2/3 of its available storage capacity. (2) *Every* I/O operation must access physical bits equivalent to 50% more than its logical size, slowing down read and write response times.

Moreover, to accommodate such an application, the SSD manufacturer would have to modify its unit of internal operations to be larger than the logical page size. Alternatively, if unmodified hardware is used, each I/O operation would have to access two physical pages, increasing its response time overhead to 100%.

The limitations of practical WOM codes complicate things even further. WOM code constructions that achieve a capacity overhead close to the theoretical lower bound ("capacity achieving") entail encoding and decoding complexities that are far from practical [51, 56]. Alternatively, more efficient constructions achieve similar capacity overhead but do not necessarily succeed in successive writes for all data combinations [6]. In other words, each such code is characterized by a small (nonnegligible) probability of failure in writing.

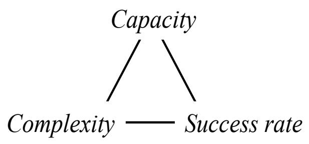


Figure 3: WOM code design space trades off storage capacity, encoding complexity (efficiency) and the rate of successful encoding.

Figure 3 depicts the inherent tradeoff of WOM code design space. Of the three objectives: capacity, complexity, and high encoding success rate, any two can be optimized at the cost of compromising the third.

3 Implementing Second Writes

Our design is based on choosing WOM code constructions suitable for real systems. We narrow our choice of WOM code by means of two initial requirements:

1. First writes must not be modified. Their encoding and data size must remain unchanged.
2. The complexity of the chosen code must not exceed that of commonly used error correction codes.

The first requirement ensures that the latency and storage utilization of most of the I/O operations performed on the SSD will remain unaffected. The second requirement enables us to parallelize or even combine WOM and ECC encoding and decoding within the SSD controller, without incurring additional delays [25].

Thus, we limit our choice to codes that satisfy the above constraints and vary in the tradeoff between storage capac-

	a	b	c	d	e	f
Req. storage	200%	206%	208%	210%	212%	214%
Success rate	0	5%	58%	95%	99%	100%

Table 2: Sample WOM codes and their characteristics. Success rates were verified in simulations on random data, as described in [6], assuming a 4KB page size. The required storage is relative to the logical page size.

ity and success rate. One such example are *polar WOM codes* [6], based on a family of error-correcting codes recently proposed by Arikan [2]. Their encoding and decoding complexities are the same as those of LDPC error correction codes [3, 6, 50]. Polar WOM codes can be constructed for all achievable capacity overheads, but with nonnegligible failure probability [6]. Table 2 summarizes several known instances which match our requirements.

The tradeoff between storage efficiency and success rate is evident. While choosing a code that always succeeds (Table 2(f)) is appealing, it requires programming three physical pages for writing a single logical page, which, on most flash architectures, cannot be done concurrently. However, a code that requires exactly two physical pages for a second write (Table 2(a)) always fails in practice.

We compromise these two conflicting objectives by utilizing the page spare area. Recall that the spare area is mainly used for error correction. However, since the bit error rates increase with block lifetime, weaker ECC can sometimes be used, utilizing only a portion of the page spare area and improving encoding performance [7, 28, 33, 44, 48, 58, 60]. We take advantage of the page spare area and divide it into two sections, as depicted in Figure 4. In the first, smaller section, we store the ECC of the first write. In the second, larger section, we store the ECC of the second write *combined* with some of the encoded data of the second write. A way to produce this combined output has been suggested in [25].

By limiting the size of the ECC of the first write, we limit its strength. Consequently, when the blocks' BER increases beyond the error-correcting capabilities of the new ECC, we must disable second writes, and the SSD operates normally with first writes only. Bit errors are continuously monitored during both reading and writing, to identify bad pages and blocks [27]. The same information can be used to determine the time for disabling second writes on each block. Another consequence of our choice of code is that WOM computations will fail with a small probability. When that happens, we simply retry the encoding; if that fails, we write the logical data as a first write. We explain this process in detail in Section 5.6, and evaluate its effect on performance in Section 6.6.

In our implementation, we use the code from Table 2(d), which requires 210% storage capacity and succeeds with a probability of 95%. We assume each page is augmented with a spare area 9% of its size [60], and allocate 2.5% for storing the ECC of the first write (Figure 4(a)). The

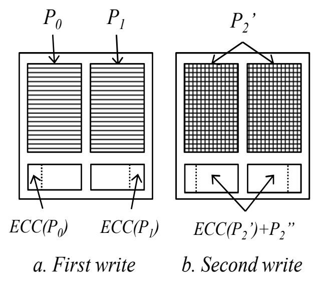


Figure 4: Use of page spare area for second writes. A small section is used for the ECC of the first write (a). The remaining area is used for the combined WOM code and ECC of the second write (b).

remaining 6.5% stores 5% of the WOM code output combined with the ECC equivalent to an ECC of 4% of the page size (Figure 4(b)). Altogether, the two physical pages along with their spare areas provide a total capacity equivalent to 210% of the logical page size. An ECC of 2.5% of the page size is sufficient for roughly the first 30% of the block's lifetime [7, 33, 48], after which we disable second writes.

The utilization of the spare area can change in future implementations, in order to trade off storage capacity and success rate, according to the size of the page spare area and the available codes. According to current manufacturing trends, BERs increase, requiring stronger ECCs, and, respectively, larger page spare area. However, the size of the spare area is set to accommodate error correction for the highest expected BER, observed as flash cells reach their lifetime limit. The Retention Aware FTL [33] combines two types of ECC, using the weaker code to improve write performance when lower BERs are expected. The same method can be used to utilize the redundant spare area for second writes.

WOM codes require that there be enough ($\geq 50\%$) zero bits on the physical page in order to apply a second write. Thus, we ensure that no more than half the cells are programmed in the first write. If a first write data page has too many one bits, we program its complement on the physical page, and use one extra bit to flag this modification. The overhead of this process is negligible [10]. The application of WOM encoding to SLC flash, where each cell represents one bit, is straightforward. In MLC and TLC flash, the cell voltage levels are mapped to four and eight possible 2-bit and 3-bit values, respectively. WOM encoding ensures that the cell level can only increase in the second write, assuming the number of one bits in each level is greater than or equal to the number of ones in all lower levels. Due to inter-cell interference, the failure probability may be higher with MLC and TLC [26].

Expected benefit. To estimate the expected reduction in the number of erasures, we perform the following best case analysis. We refer to an SSD that performs only first writes as a *standard SSD*. Assume that each block contains N pages, and that there are M page write requests. The expected number of erasures in a standard SSD is $E = \frac{M}{N}$. In Reusable SSD, $N + \frac{N}{2}$ pages can be written on each block before it is erased. Thus, the expected number

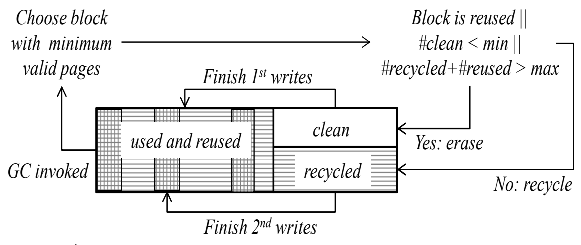

Choose block with minimum valid pages
Block is reused || #clean < min || #recycled+#reused > max
Finish 1st writes
used and reused
clean
recycled
Yes: erase
GC invoked
No: recycle
Finish 2nd writes

Figure 5: Garbage collection and block lifecycle

of erasures is $E' = \frac{M}{N+N/2} = \frac{2}{3}E$, a reduction of 33% compared to a standard SSD.

To calculate the effect on the SSD's lifetime, consider a standard SSD that can accommodate W page writes. 30% of them ($0.3W$) are written in the first 30% of the SSD's lifetime. With second writes, 50% more pages can be written ($0.15W$), resulting in a total of $1.15W$, equivalent to an increase of 15% in the SSD's lifetime.

4 Reusable SSD: Overview

In our design, blocks transition between four states, as depicted in Figure 5. In the initial, *clean* state, a block has never been written to, or has been erased and not written to since. A block moves to the *used* state after all of its pages have been written by a first write. When a used block is chosen as victim by the garbage collection process, it is usually recycled, moving to the *recycled* state, which means its invalid pages can be used for second writes. When all of the invalid pages on a recycled block have been written, it moves to the *reused* state. When a reused block is chosen by the garbage collection process, it is erased, moving back to the clean state.

Note that a used block can, alternatively, be erased, in which case it skips the recycled and reused states, and moves directly to the clean state. The garbage collection process determines, according to the number of blocks in each state, whether to erase or recycle the victim block. Figure 5 provides a high level depiction of this process, explained in detail in Section 5.4.

Figures 5 and 6 show how blocks are organized within the two planes in a flash chip. We divide the physical blocks in each plane into three logical partitions: one for clean blocks, one for recycled ones, and one for used and reused blocks. One clean block and one recycled block in each plane are designated as $CleanActive$ and $RecycledActive$, respectively. First writes of pages that are mapped to a specific flash chip are performed, independently, on any of the two $CleanActive$ blocks in that chip. Second writes are performed in parallel on both $RecycledActive$ blocks in the chip.

A logical page is written as a second write if (1) recycled blocks are available for second writes in both planes, and (2) the data written has been classified as hot. Pages written as first writes are divided between planes to balance the number of valid pages between them. Figure 6

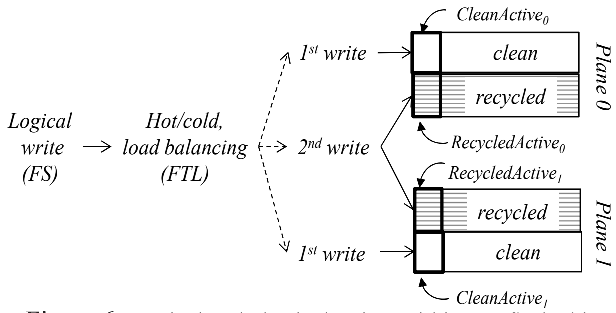


Figure 6: Logical and physical writes within one flash chip

provides a high level description of this process. A detailed description of our design is given in the next section.

5 Design Details

5.1 Page Allocation

Within an SSD, logical data is striped between several chips. The *page allocation scheme* determines, within each flash chip, to which plane to direct a logical write request. The target plane need not be the one on which the previous copy of the page was written. The standard scheme, which we modify for second writes, balances the number of clean pages in each plane [1]. Thus, a write request is directed to the plane that currently has fewer clean pages than the other.

We adapt the standard scheme to second writes as follows. When a page is classified as hot, it is written in parallel to a pair of $RecycledActive$ blocks, one in each plane, as depicted in Figure 6. To minimize the size of additional metadata, we require that a second write be performed on a pair of pages with identical offset within their blocks. Thus, we maintain an offset counter, advanced after each second write, that points to the minimal page offset that corresponds to invalid data in both $RecycledActive$ blocks. The two pages are written in parallel, utilizing the specific architecture's set of parallel or multiplane commands.

The modification to the page allocation scheme is minimal. First writes are divided between planes as before. Read requests of pages written in first writes are served as before. Read requests of pages written in second writes are served in parallel, using the respective parallel read command.

Our requirement that second write pages have identical offset affects performance only slightly. Although invalid pages may be dispersed differently in each $RecycledActive$ block, this limitation is negligible in practice. Most blocks are recycled with a very small valid count (up to 7% of the block size in our experiments), so most invalid pages can be easily reused.

5.2 Page Mapping

Our design is based on a page mapping FTL, which maintains a full map of logical pages to physical ones in the *page map* table. Since every logical page may be mapped to two physical pages, the page map in a naive implementation would have to double in size. However, the size of the page map is typically already too large to fit entirely in memory. Thus, we wish to limit the size of the additional mapping information required for second writes.

To do so, we require that in a second write, the logical page be written on two physical pages with identical offset within the two $RecycledActive$ blocks. We maintain a separate mapping of blocks and their pairs, in a table called the *block map*, while the page map remains unmodified. Each block map entry corresponds to a block in $plane0$, and points to the pair of this block in $plane1$ on the same chip. Entries corresponding to clean and used blocks are null.

For a page written in a first write, the page map points to the physical location of this page. For a page written in a second write, the map points to the physical location of the first half of this page, in $plane0$. Thus, if the page map points to a page in $plane0$ and the corresponding block map entry is non-null, the page is stored on two physical pages, whose addresses we now know.

For a rough estimate of the block map size, assume that 2 byte block addresses are used — enough to address 64K blocks. The block map maintains entries only for blocks in $plane0$, so for a drive with 64K blocks we would need 64KB for the block map. Such a drive corresponds to a logical capacity of 16GB to 128GB, with blocks of size 256KB [55] to 2MB [19], respectively. 64KB is very small compared to the size of page mapping FTLs, and to the available RAM of modern SSDs [12]. Thus, the overhead required for mapping of second writes is negligible. The block map can be further compacted, if, instead of storing full block addresses, it would store only the relative block address within $plane1$.

The state-of-the-art page mapping FTLs, such as DFTL [15], add a level of indirection to selectively cache the hottest portion of the page map. Within such FTLs, the block map necessary for second writes can be used in a similar manner. However, due to its small size, we can reasonably assume that the block map will be stored entirely in memory, without incurring any additional lookup and update overhead.

Hybrid or block mapping FTLs typically use page-mapped *log blocks* on which data is initially written, before it becomes cold and is transferred for long term storage on *data blocks* [9]. These FTLs can be modified as described above, to apply second writes to pairs of log blocks. Although data blocks can also be paired for second writes, it may be advisable to restrict second writes only to the hot data in the log.

5.3 Prefetching Invalid Pages

Recall that a WOM encoding requires the invalidated data currently present on the two physical pages. Thus, one page must be read from each $RecycledActive$ block in a chip, before a second write can be performed. In our design, second writes are always directed to the next pair of invalid pages available on the current pair of $RecycledActive$ blocks. Thus, these pages can be *prefetched* — read and stored in the SSD's RAM, as soon as the previous second write completes.

One page must be prefetched for each plane. Thus, for a typical architecture of 8KB pages, 1MB of RAM can accommodate prefetching for 128 planes, equivalent to 64 flash chips. In the best case, prefetching can completely eliminate the read overhead of second writes. In the worst case, however, it may not complete before the write request arrives, or, even worse, delay application reads or other writes. Our experiments, described in Section 6.4, show that the latter is rare in practice, and that prefetching significantly reduces the overall I/O response time.

5.4 Garbage Collection and Recycling

We modify the standard garbage collection process to handle block recycles. Recall that greedy garbage collection always picks the block with the minimum valid count as victim for cleaning and erasure. Ideally, using second writes, every used block would first be recycled and reused before it is erased. However, our goal of preserving the exported storage capacity and performance of the SSD imposes two restrictions on recycling.

Minimum number of clean blocks. When a victim block is cleaned before erasure, its valid pages are *moved*: they are invalidated and copied to the active block. We require that valid pages move to $CleanActive$, and not to $RecycledActive$, for two reasons. First, to avoid dependency in the cleaning process, so that cleaning in both planes can carry on concurrently, and second, so that remaining valid pages that are likely cold will be stored efficiently by first writes. Thus, at least two clean blocks must be available in each plane for efficient garbage collection.

Maximum number of reused and recycled blocks. To preserve the drive's exported logical size, we utilize its overprovisioned space for second writes as follows. Consider a drive with T physical blocks and U logical blocks, resulting in an overprovisioning ratio of $\frac{T-U}{U}$. Then physical pages with capacity equivalent to $R = T - U$ blocks are either clean, or hold invalid data. For second writes, we require that the number of blocks in the recycled or reused states not exceed $2R$. Since second writes occupy twice the capacity of first writes, this number of blocks can store a logical capacity equivalent to R. Thus, the drive's physical capacity is divided between $T - 2R$ blocks holding first writes, and $2R$ blocks holding data of size R in second writes, with a total logical capacity of $T - 2R + R$, equivalent to the original logical capacity, U.

Garbage collection is invoked when the number of available clean blocks reaches a given $threshold$. We modify the standard greedy garbage collector to count recycled blocks towards that threshold. When the threshold is reached in a plane, the block with the minimum number of valid pages in this plane is chosen as victim. The block is erased if (1) it is reused, (2) there are fewer than 2 clean blocks in the plane, or (3) the total of reused and recycled blocks is greater than $2R$. Otherwise, the block is recycled (see Figure 5).

While wear leveling is not an explicit objective of our design, blocks effectively alternate between the 'cold partition' of first writes, and the 'hot partition' of second writes. Further wear leveling optimizations, such as retirement, migrations [1] and fine grained partitioning [54], are orthogonal to our design, and can be applied at the time of block erasure and allocation.

Cleaning reused blocks. Special consideration must be given to second write pages that are still valid when a reused block is recycled. Usually, each plane is an independent *block allocation pool*, meaning garbage collection invocation, operation, and limits apply separately to each plane. This allows page movement during cleaning to be performed by *copyback*, transferring data between blocks in the same plane, avoiding inter-plane bus overheads.

Each reused block chosen as victim in one plane has a pair in the second plane of that chip. However, since each block may also have valid pages of first writes, a block may be chosen as victim while its pair does not have the minimum valid count in its plane. Thus, we clean only the victim block, as follows. All valid pages of first writes are moved as usual. Valid pages of second writes are read from both blocks, and must be transferred all the way to the controller for WOM decoding[2]. Then they are written as first writes in the plane on which garbage collection was invoked. The overhead of this extra transfer and decoding is usually small, since most reused blocks are cleaned with a low valid count (usually below 7% of the block size).

5.5 Separating Hot and Cold Data

The advantages of separating hot and cold data have been evaluated in several studies [9, 13, 20, 27, 38, 42, 54]. In our design, this separation is also motivated by the need to maintain the drive's original capacity and performance. The largest benefit from second writes is achieved if they are used to write hot data, as we explain below.

When a reused block is cleaned before erasure, all remaining valid pages must be copied elsewhere. Second write pages that are moved to the active block are not "free," in the sense that they end up generating first writes. If second writes are used only for hot data, we can expect it to be invalidated by the time the block is chosen for cleaning.

[2]Recent SSDs require similar transfer for valid pages of first writes, for recalculation of the ECC due to accumulated bit errors.

In addition, in order to maximize the potential of second writes, we wish to avoid as much as possible cases in which used blocks are erased without being reused. This may happen if too many reused blocks have a high valid page count, and the number of reused and recycled blocks reaches $2R$. Then, the garbage collector must choose used blocks as victims and erase them.

The use of a specific hot/cold data classification scheme is orthogonal to the design of Reusable SSD. As a proof of concept, we identify hot/cold data according to the size of its file system I/O request. It has been suggested [9, 20] that large request sizes indicate cold data. We classify a logical page as cold if its original request size is 64KB or more. We also assume, as in previous work [20, 42], that pages that are still valid on a block chosen by the garbage collector are cold. Thus, pages moved from a block before it is erased are also classified as cold. Cold data is written as first writes, and hot data as second writes, if recycled blocks are available (see Figure 6).

5.6 Handling Second Write Failures

Our design uses WOM codes that fail with a nonnegligible probability (the success rate is $P = 95\%$ in our implementation). A failed encoding means that the output contains 0 bits in places corresponding to cells in the invalidated pages that have already been programmed to 1.

The simplest approach to handling such failures is to simply abort the second write, and write the data on a clean block as a first write. The first write requires additional latency for computing the ECC, typically 8us [60], but is guaranteed to succeed. Within our design, choosing this approach would imply that 5% of the hot pages destined for second writes end up occupying pages in cold blocks.

A different approach is to handle the problematic bits in the same manner as bit errors in standard first writes. The output is programmed on the physical pages as is, and the ECC 'fixes' the erroneous bits. However, recall that we already 'sacrificed' some ECC strength for implementing second writes, and the number of erroneous bits may exceed the remaining error-correction capability.

Our approach is to *retry* the encoding, i.e., recompute the WOM code output. In the general case, this can be done by encoding the logical data for writing on an alternative pair of invalid pages. The two attempts are independent in terms of success probability, because they are applied to different data combinations. Thus, the probability of success in the first encoding *or* the retry is $P' = 1 - (1 - P)^2$, or 99.75% in our case. This value is sufficient for all practical purposes, as supported by our evaluation in Section 6.6.

The overhead incurred by each retry is that of the additional WOM computation, plus that of reading another pair of physical pages. However when using Polar WOM codes, the extra read overhead can be avoided. Due to the probabilistic nature of these codes, the retry can be performed using the *same* physical invalidated data, while only altering an internal encoding parameter [6]. Successive retries are independent, yielding a similar overall success probability as with the general retry method described above. Thus, in our design, a WOM code failure triggers one retry, without incurring an additional read. If the retry fails, the page is written as a first write. We evaluate the overhead incurred by retries in both the special and general cases in Section 6.6.

6 Evaluation

We performed a series of trace driven simulations to verify that second writes in Reusable SSD are indeed 'free.' We answer the following questions. (1) How many erasures can be eliminated by second writes? (2) What is the effect of second writes on read and write latency? (3) Do second writes incur additional overheads? (4) How sensitive are the answers to design and system parameters? In addition, we establish the importance of our main design components.

6.1 Experimental Setup

We implemented second writes within the MSR SSD extension [1] of DiskSim [5]. We configured the simulator with two planes in each flash chip, so that each plane is an independent allocation pool, as described in Section 5.4. We allow for parallel execution of commands in separate planes within each chip. Second writes are simulated by mapping a logical page to two physical pages that have to be read and written. We use a random number generator to simulate encoding failures, and disable second writes on blocks that reach 30% of their lifetime.

The SSD extension of DiskSim implements a greedy garbage collector with wear leveling and migrations, copying cold data into blocks with remaining lifetime lower than a threshold. We modify this process as described in Section 5.4, so that it applies only to victim blocks that are going to be erased. Garbage collection is invoked when the total of clean *and* recycled blocks in the plane drops below a threshold of 1%. DiskSim initializes the SSD as full. Thus, every write request in the trace generates an invalidation and an out-of-place write. We use two common overprovisioning values, 7% and 28%, which represent consumer and enterprise products, respectively [52]. We refer to the unmodified version of DiskSim, with first writes only, as the *standard SSD*.

We evaluate our design using real world traces from two sources. The first is the MSR Cambridge workload [40], which contains traces from 36 volumes on 13 servers. The second is the Microsoft Exchange workload [41], from one of the servers responsible for Microsoft employee e-mail. The volumes are configured as RAID-1 or RAID-5 arrays, so some of them are too big to fit on a single SSD.

Volume	Requests (M)	Drive size (GB)	Requests/sec	Peak writes/sec	Write ratio	Average write size (KB)	Total writes (GB)
zipf(1,2)	3	4	200	200	1	4	12
src1_2	2	16	3.15	95.69	0.75	33	45
stg_0	2		3.36	10.23	0.85	10	16
hm_0	4	32	6.6	48.62	0.64	9	23
rsrch_0	1.5		2.37	6.16	0.91	9	11
src2_0	1.5		2.58	19.95	0.89	8	10
ts_0	2		2.98	14.4	0.82	8	12
usr_0	2.5		3.7	12.54	0.6	11	14
wdev_0	1		1.89	7.41	0.8	8	7
prxy_0	12.5	64	20.7	42.57	0.97	7	83
mds_0	1		2	5.61	0.88	8	8
proj_0	4		6.98	132.71	0.88	41	145
web_0	2		3.36	18.52	0.7	13	17
prn_0	5.5	128	9.24	145.1	0.89	11	54
exch_0	4		45.28	90.56	0.92	27	94
src2_2	1	256	1.91	271.41	0.7	55	42
prxy_1	24		278.83	120.81	0.35	13	106

Table 3: Trace characteristics. The duration of all production traces is one week, except prxy_1 and exch_0, which are one day.

Manufacturer	Type	Pages/ Block	Read (us)	Write (ms)	Erase (ms)	Size (Gb)
Toshiba [55]	SLC	64	30	0.3	3	32
Samsung [49]	MLC	128	200	1.3	1.5	16
Hynix [19]	MLC	256	80	1.5	5	32

Table 4: NAND flash characteristics used in our experiments.

We used the 16 traces whose address space could fit in an SSD size of 256GB or less, and that included enough write requests to invoke the garbage collector on that drive. These traces vary in a wide range of parameters, summarized in Table 3. We also used two synthetic workloads with Zipf distribution, with exponential parameter $\alpha = 1$ and 2. Note that a perfectly uniform workload is unsuitable for the evaluation of second writes, because all the data is essentially cold.

We use parameters from 3 different NAND flash manufacturers, corresponding to a wide range of block sizes and latencies, specified in Table 4. While the flash packages are distributed in different sizes, we assume they can be used to construct SSDs with the various sizes required by our workloads. Due to alignment constraints of DiskSim, we set the page size to 4KB for all drives. To maintain the same degree of parallelism for all equal sized drives, we assume each chip contains 1GB, divided into two planes. The number of blocks in each plane varies from 512 to 2048, according to the block size. The MSR SSD extension implements one channel for the entire SSD, and one data path (way) for each chip. We vary the number of chips to obtain the drive sizes specified in Table 3.

6.2 The Benefit of Second Writes

Write amplification is commonly used to evaluate FTL performance, but is not applicable to our design. Second writes incur twice as many physical writes as first writes but these writes are performed after the block's capacity has been exhausted by first writes, and do not incur additional erasures. Thus, to evaluate the performance of Reusable SSD, we measure the relative number of erasures and relative response time of our design, compared to the standard SSD. Note that in a standard SSD, the number of erasures is an equivalent measure to write amplification. In all our figures, the traces are ordered by the amount of data written compared to the physical drive size, i.e., in mds_0 the least data was written, and in zipf (1) and (2) the most.

Erasures. Figure 7 shows the relative number of erasures of Reusable SSD compared to the standard SSD. Recall that according to the best case analysis, Reusable SSD can write up to 50% more pages on a block before its erasure, corresponding to a reduction of 33% in the number of block erasures. In most traces, the reduction is even slightly better, around 40%. This is due to the finite nature of our simulation – some of the recycled blocks were not erased within the duration of the simulation. Since Reusable SSD can apply second writes in the first 30% of the drive's lifetime, it performs additional writes equivalent to a lifetime extension of 15%.

In several traces the reduction was less than 33%, because a large portion of the data was written in I/O requests larger than the 64KB threshold. The corresponding pages were classified, mostly correctly, as cold and written as first writes, so the full potential of second writes was not realized. In traces src2_2, prxy_1, exch_0, prxy_0, proj_0 and src1_2, the percentage of such cold pages was 95, 44, 83, 32, 86 and 78, respectively (compared to 1%-15% in the rest of the traces). We further investigate the effect of the hot/cold threshold in Section 6.5.

The different block sizes of the drives we used affect the *absolute* number of erasures, both for standard and for Reusable SSD. However, the *relative* number of erasures was almost identical for all block sizes.

The synthetic Zipf workloads have no temporal locality, so more pages remain valid when blocks are erased or recycled, especially with zipf(1) which has less access skew than zipf(2). With low overprovisioning, Reusable SSD is less efficient in reducing erasures for this workload because there are fewer invalid pages for use in second writes.

The reduction in the number of erasures usually depends on the drive's overprovisioning (OP). Higher overprovisioning means the maximum number of blocks that can be in the reused or recycled states ($2R$) is higher, thus allowing more pages to be written in second writes. In the extreme case of trace src2_2 with OP=28% , all writes could be accommodated by the overprovisioned and recycled blocks, thus reducing the number of erasures to 0. In the other traces with a high percentage of cold data, the number of erasures did not decrease further with overprovisioning because fewer blocks were required to accommodate the "hot" portion of the data. We examine a wider range of overprovisioning values in Section 6.6.

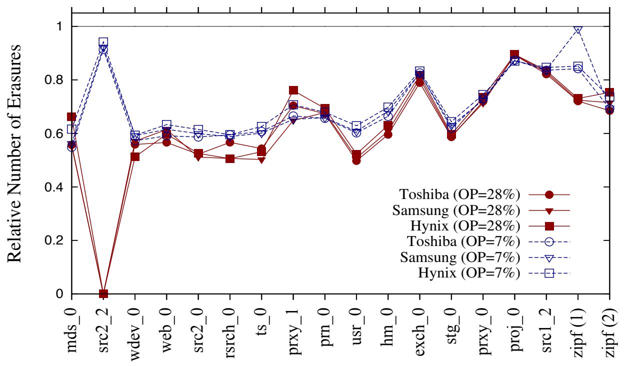


Figure 7: Relative number of erasures of Reusable SSD compared to the standard SSD. The reduction in erasures is close to the expected 33% in most cases.

Performance. Overprovisioning notably affects the performance of first as well as second writes. When overprovisioning is low, garbage collection is less efficient because blocks are chosen for cleaning while they still have many valid pages that must be moved. This degrades performance in two ways. First, more block erasures are required because erasures do not generate full clean blocks. Second, cleaning before erasure takes longer, because the valid pages must be written first. This is the notorious delay caused by write amplification, which is known to increase as overprovisioning decreases.

Indeed, the reduction in erasures achieved by Reusable SSD further speeds up I/O response time when overprovisioning is low. Figure 8 shows the reduction in average I/O response time achieved by Reusable SSD compared to the standard SSD. I/O response time decreased by as much as 15% and 35%, with OP=28% and OP=7%, respectively.

The delay caused by garbage collection strongly depends on write and erase latencies, as well as on the block size. When overprovisioning is low (7%) and writes cause a major delay before erasures, the benefit from second writes is greater for drives with longer write latencies – the benefit in the Hynix setup is up to 60% greater than in the Toshiba setup. When overprovisioning is high (28%) and the cost of cleaning is only that of erasures, the benefit from second writes is greater for drives with small blocks whose absolute number of erasures is greater – the benefit in the Toshiba setup is up to 350% greater than the benefit in the Hynix setup.

6.3 The Benefit of Parallel Execution

To establish the importance of parallelizing second writes, we implemented a "sequential" version of our design, where second writes are performed on a pair of contiguous invalid pages on the same block. The two planes in each chip can still be accessed concurrently – they each have an independently written $RecycledActive$ block.

The reduction in the number of cleans is almost iden-

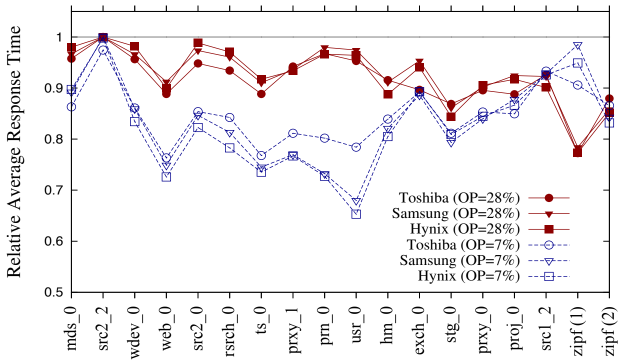


Figure 8: Relative I/O response time of Reusable SSD compared to the standard SSD. The reduction in erasures reduces I/O response time, more so with lower overprovisioning.

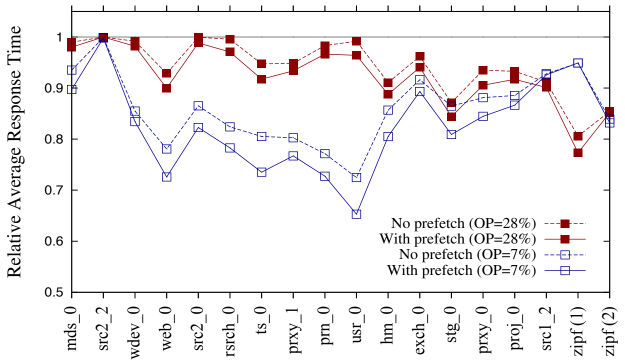


Figure 9: Relative I/O response time of Reusable SSD with and without prefetching, in the Hynix setup. Prefetching always reduces I/O response time.

tical to that of the parallel implementation, but the I/O response time increases substantially. In the majority of traces and setups, second writes increased the I/O response time, by an average of 15% and as much as 31%. We expected such an increase. Data written by a second write requires twice as many pages to be accessed, both for reading and for writing, and roughly 33% of the data in each trace is written as second writes.

As a matter of fact, the I/O response time increased less than we expected, and sometimes *decreased* even with the sequential implementation. The major reason is the reduction in erasures – the time saved masks some of the extra latency of second writes. Another reason is that although roughly 33% of the data was written in second writes, only 1%-19% of the reads (2%-6% in most traces) accessed pages written in second writes. This corresponds to a well-known characteristic of secondary storage, where hot data is often overwritten without first being read [53].

Nevertheless, an increase of 15%-30% in average response time is an unacceptable performance penalty. Our parallel design complements the reduction in erasures with a significant reduction in I/O response time.

6.4 The Benefits of Prefetching Invalid Pages

To evaluate the contribution of prefetching invalid pages, we disabled prefetching and repeated our experiments. Figure 9 shows the results for the Hynix setup with OP=28% and OP=7%. These are the two setups where second writes achieved the least and most reduction in I/O response time, respectively. These are also the setups where the contribution of prefetching was the highest and lowest, respectively.

With OP=7%, and specifically the Hynix setup, the reduction in erasures was so great that the extra reads before second writes had little effect on overall performance. Prefetching reduced I/O response time by an additional 68% at most. With OP=28%, where the reduction in I/O response time was less substantial, prefetching played a more important role, reducing I/O response time by as much as $\times 21$ more than second writes without it.

The results for the rest of the setups were within this range; the average I/O response time for second writes with prefetching was 120% shorter than without it. Prefetching never delayed reads or first writes to the point of degrading performance.

6.5 The Benefits of Hot/Cold Classification

When an I/O request size is equal to or larger than the hot/cold threshold, its data is classified as cold and written in first writes. We examine the importance of separating hot data from cold, and evaluate the sensitivity of our design to the value of the threshold. We varied the threshold from 16KB to 256KB. Figure 10 shows the results for the Toshiba setup – the results were similar for all drives. We present only the traces for which varying the threshold changed the I/O response time or the number of erasures. The results for 256KB were the same as for 128KB.

Figure 10(a) shows that as the threshold increases, more data is written in second writes, and the reduction in the number of erasures approaches the expected 33%. However, increasing the threshold too much sometimes incurs additional cleans. For example, in prn_0, data written in requests of 64KB or larger nearly doubled the valid count of victim blocks chosen for cleaning, incurring additional delays as well as additional erase operations. Figure 10(c) shows that a reduction [increase] in the number of erasures due to higher thresholds entails a reduction [increase] in the relative I/O response time.

Figures 10(b) and 10(d) show the results for the same experiments with OP=28%. The additional overprovisioned capacity extends the time between cleans, to the point where even the cold data is already invalid by the time its block is erased. Both the number of erasures and the I/O response time decrease as more data can be written in second writes. Specifically, Figure 10(b) shows that the full "50% free" writes can be achieved in enterprise class setups. Still, the hot/cold classification guarantees better performance, possibly at the price of limiting the reduction in erasures.

An adaptive scheme can set the threshold according to

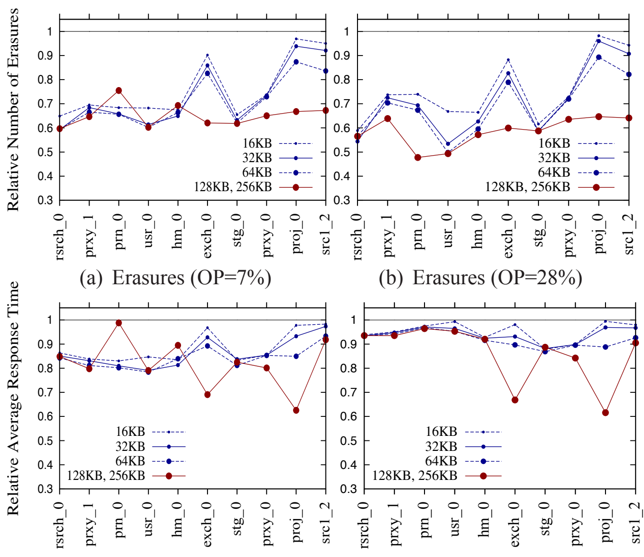


Figure 10: Relative I/O response time and erasures when varying the hot/cold classification threshold. Increasing the threshold too much might increase I/O response time.

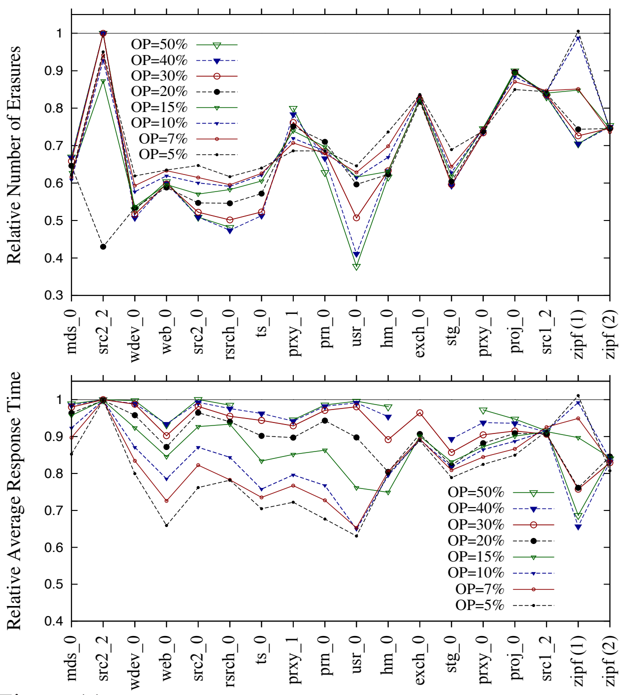


Figure 11: Relative number of erasures (top) and average I/O response time (bottom) with the Hynix setup, with varying overprovisioning ratios.

the observed workload to optimize both objectives, but is outside the scope of this work. Alternatively, classification can be done using recent optimized schemes (see Section 7) for more accurate results. Note that regardless of the classification scheme, Reusable SSD also separates application writes from garbage collection writes. This separation is expected to reduce the number of erasures compared to the standard SSD, even without second writes.

6.6 Sensitivity Analysis

Overprovisioning. For a comprehensive analysis we repeated our experiments, varying the overprovisioning value from 5% to 50%[3]. For all the drives and traces, the number of erasures and the I/O response time decreased as overprovisioning increased, both in the standard SSD and in Reusable SSD. Figure 11 shows the relative number of erasures and relative I/O response time of Reusable SSD compared to the standard SSD. We show results for the Hynix setup, where varying the overprovisioning value had the largest effect on these two measures.

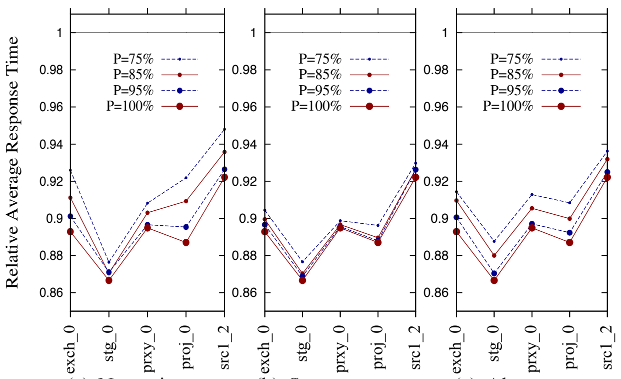
These results support our observation in Section 6.2, that the reduction in erasures is larger when overprovisioning is higher, except in traces that have a high portion of cold data written as first writes. Reusable SSD reduces I/O response time more with lower overprovisioning, where erasures cause longer delays. The maximal variation in relative average response time was 24%, 32%, and 35% in the Toshiba, Samsung and Hynix setups, respectively.

WOM encoding failures. Reusable SSD is designed to work with any WOM code construction that satisfies the requirements specified in Section 3. To evaluate the sensitivity of our design to WOM code characteristics, we repeated our experiments, varying the encoding success rate from 75% to 100%.

In the first set of experiments we disable retries completely, so they serve as a *worst case* analysis. On a WOM encoding failure we default to a first write on the $CleanActive$ block. Every such failure incurs the overhead of an additional ECC computation, because ECC must be computed for the logical data. The ECC for a 4KB page can usually be computed in less than $10\mu s$ [60]. To account for changes in page size, ECC and WOM code, and as a worst case analysis, we set the overhead to half the read access time in each drive.

Figure 12(a) shows the relative I/O response time of Reusable SSD without retries, compared to the standard SSD. Surprisingly, varying the success rate resulted in a difference in relative I/O response time of less than 1% for all traces with OP=7%, and for most traces with OP=28%. The reduction in erase operations was not affected at all. We show here only the traces for which the difference was larger than 1%. We show the results with the Toshiba setup because the differences with the other setups were even smaller. The reason for such small differences is that in most traces, the maximum allowed number of reused and recycled blocks does not accommodate all the hot data, and some hot data is written as first writes when no recycled block is available. Thus, WOM encoding failures

[3]The address space of ts_0, exch_0 and stg_0 was too large to fit in the respective drive sizes from Table 3 with OP=50% (and OP=40% for exch_0). Thus, the data points corresponding to those traces and OP values are missing.



(a) No retries (b) Same pages (c) Alt. pages

Figure 12: The sensitivity of Reusable SSD to varying WOM encoding success rates with no retries (a), retries on the same physical pages (b) and retries on alternative physical pages (c), with the Toshiba setup and OP=28%.

simply distribute the hot data differently, incurring only the additional ECC computation delay.

Figure 12(b) shows the relative I/O response time of Reusable SSD with retries, as described in Section 5.6. A retry incurs the same overhead as the first WOM encoding. If that retry fails, extra overhead is incurred by the ECC computation of the first write. Note that with one retry, the overall probability of successful encoding with $P = 75\%$ is $P' = 1 - (1 - 0.75)^2 = 93.75\%$. Indeed, the performance of Reusable SSD with $P = 75\%$ is almost equal to that of Reusable SSD without retries and $P = 95\%$. Similarly, the relative I/O response time with $P = 95\%$ and one retry is almost equal to that of using a WOM code with no encoding failures ($P = 100\%$).

We also examine the applicability of our design to WOM codes that do not guarantee independent success probability on the same invalid data. Thus, we ran one more set of experiments where, upon a WOM encoding failure, an additional pair of invalid pages is read, and the encoding is retried on these pages. In this variation, the overhead of retrying is the same as in our design, plus an additional read latency. Our results, presented in Figure 12(c), show that the additional overhead is negligible when P is high (95%), but nonnegligible for smaller values of P ($\leq 85\%$). In addition, unlike the first two approaches, this overhead appeared in the rest of the traces as well, and also with OP=7%. Still, even in those cases, the I/O response times were reduced substantially compared to the standard SSD, and the difference between $P = 100\%$ and $P = 75\%$ was always below 4%.

Energy consumption. According to a recent study [39], the energy consumption of an erase operation is one order of magnitude larger than that of a write operation, but the energy it consumes per page is the smallest of all operations. Of all measured operations, writes are the major contributor to energy consumption in flash chips. In Reusable SSD, roughly 33% of the pages are written in second writes, programming two pages instead of one. Thus, one might expect its energy consumption to increase in proportion to the increase in physical writes.

However, this same study also showed the energy consumed by writes to depend on the number of programmed cells. But not only do second writes not require programming twice as many cells, their overall number of programmed bits is expected to equal that of first writes [6]. We thus expect the energy consumption to decrease, thanks to the reduction in erasures that comes with second writes. Measurements on a naive implementation of second writes showed such a reduction [14], and we believe these results will hold for Reusable SSD. A more accurate evaluation remains part of our future work.

7 Related Work

The Flash Translation Layer is a good candidate for manipulating flash traffic to extend SSD lifetime. Most FTLs implement some notion of wear leveling, where cold data is migrated to retired or about-to-be-retired blocks, and blocks are allocated for writing according to their erase count or wear [1, 20, 24, 27, 28, 38, 42]. Buffering [29] and even deduplication [16] are used by some FTLs to reduce the number of flash writes.

Another approach reduces write traffic to the SSD by eliminating write operations at higher levels of the storage hierarchy. Such methods include a hard disk based write cache [53], specialized file systems and data bases [11, 31, 34, 36, 38], and admission control in flash based caches [18, 43, 46].

A recent analytic study showed that separating hot and cold data can minimize write amplification so that it approaches 1 [13]. Indeed, many FTLs write hot and cold data into separate partitions [9, 20, 27, 28, 38, 42, 54]. They classify hot data according to I/O request size [9, 20], time and frequency of write [38, 54], and whether the write was generated by the garbage collector [42].

Reusable SSD separates hot and cold data, and applies wear leveling and migration to blocks that are about to be erased. However, the specific classification or wear leveling technique is orthogonal to our design, and can be replaced with any of the state-of-the-art methods to combine their advantages with those of second writes. Similarly, when some of the write traffic is eliminated from the SSD, Reusable SSD can apply roughly 33% of the remaining writes to reused blocks, eliminating additional erasures.

More intrusive methods for extending SSD lifetime include modifying the voltage of write and erase operations [24], and even explicitly delaying requests to allow cell recovery [30]. They incur an overhead that limits the SSD's performance. Reusable SSD extends SSD lifetime without requiring any changes in flash hardware. More importantly, it improves — rather than degrades — performance. Still, Reusable SSD can also be combined with such methods, to further extend device lifetime.

While the number of erasures is the most commonly used measure of device lifetime, recent studies show that cell programming has a substantial impact on their wear. They show that programming MLC cells as SLC [26], or occasionally 'relieving' them from programming [27] can significantly slow down cell degradation, regardless of the number of erasures. Second writes result in a higher average voltage level of flash cells, possibly increasing their wear. Thus, with second writes, 50% more writes can be performed before each erasure, but the number of 'allowed' erasures might decrease. However, as long as the increase in cell wear is smaller than 50%, second writes extend device lifetime. Since cell degradation is not linear with average voltage level, the magnitude of this effect cannot be derived from previous studies, and remains to be verified in future work. Our analysis of the benefits of Reusable SSD is conservative, disabling second writes on all pages after 30% of the block's lifetime. A more accurate model of cell wear can facilitate additional second writes on SLC flash or on the LSB pages in MLC flash.

Several studies suggested using WOM codes to extend SSD lifetime. Their designs are all based on an increased page size, as in Figure 2, resulting in greatly reduced capacity [4, 14, 21, 35, 57]. In [42], the capacity loss is bound by limiting second writes to several blocks. The authors of [23] assume the logical data has been compressed by the upper level, to allow for the overhead of WOM encoding. None of these studies address the additional latencies of reading invalid data before encoding and of reading and writing larger pages. In addition, most of them rely on capacity achieving codes, ignoring their high complexity or their nonnegligible failure rate [35, 42]. The design of Reusable SSD addresses all the practical aspects of second writes with off-the-shelf flash products and efficient coding techniques, achieving both performance improvements and a lifetime extension of up to 15%.

The above studies use write amplification to evaluate their designs, but it is not the correct figure of merit for multiple writes. Consider a best case example where a code with minimal 29% space overhead achieves a write amplification of 1. Still, the amount of physical data written is 29% more than the logical data written by the application. Thus, for correct evaluation, the number of erasures incurred in various designs should be compared, on SSDs with the same block size and overprovisioning.

The use of WOM codes has also been suggested for extending PCM lifetime [22, 32, 59]. The corresponding studies show a reduction in energy consumption and cell wear, but sacrifice either capacity, performance, or both.

8 Conclusions and Future Directions

We presented Reusable SSD, a practical design for applying second writes to extend SSD lifetime while significantly improving performance. Our design is general and is applicable to current flash architectures, requiring only minor adjustments within the FTL, without additional hardware or interface modifications.

Nevertheless, more can be gained from Reusable SSD as technology advances in several expected directions. Flash architectures that allow for higher degrees of parallelism can accommodate third and maybe fourth writes, combining 4 and 8 physical pages per logical page, respectively [6]. As the hardware implementation of Polar WOM codes matures, its encoding overheads will decrease [3, 50], enabling faster retries, and possibly use of constructions with higher success probability. Similarly, stronger ECCs can compensate for increasing BERs, increasing the percentage of a block's lifetime in which it can be recycled before erasure.

Finally, most previously suggested schemes for extending SSD lifetime are orthogonal to the design of Reusable SSD, and can be combined with second writes. The performance improvement achieved by Reusable SSD can mask some of the overheads of those schemes that incur additional latencies.

Acknowledgments

We thank our shepherd Peter Desnoyers and the anonymous reviewers for their insightful comments that helped improve this paper. We thank Yue Li for his help with the Polar WOM code experiments, Ran Koretzki for his help with the simulation setup, and Abdalla Amaria and Muhammad Musa for their help with the traces.

References

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *USENIX Annual Technical Conf. (ATC)*, 2008.

[2] E. Arikan. Channel polarization: A method for constructing capacity-achieving codes for symmetric binary-input memoryless channels. *IEEE Trans. on Inform. Theory*, 55(7):3051–3073, 2009.

[3] A. Balatsoukas-Stimming, A. Raymond, W. J. Gross, and A. Burg. Hardware architecture for list successive cancellation decoding of polar codes. *IEEE Trans. on Circuits and Systems II: Express Briefs*, 61(8):609–613, Aug 2014.

[4] A. Berman and Y. Birk. Retired-page utilization in write-once memory – a coding perspective. In *IEEE Intl. Symp. on Inform. Theory (ISIT)*, 2013.

[5] J. S. Bucy, J. Schindler, S. W. Schlosser, and G. R. Ganger. The DiskSim simulation environment version 4.0 reference manual, May 2008.

[6] D. Burshtein and A. Strugatski. Polar write once memory codes. *IEEE Trans. on Inform. Theory*, 59(8):5088–5101, 2013.

[7] Y. Cai, E. F. Haratsch, O. Mutlu, and K. Mai. Error patterns in MLC NAND flash memory: Measurement, characterization, and analysis. In *Conf. on Design, Automation and Test in Europe (DATE)*, 2012.

[8] P. Cappelletti, R. Bez, D. Cantarelli, and L. Fratin. Failure mechanisms of flash cell in program/erase cycling. In *Intl. Electron Devices Meeting (IEDM) Technical Digest*, 1994.

[9] M.-L. Chiao and D.-W. Chang. ROSE: A novel flash translation layer for NAND flash memory based on hybrid address translation. *IEEE Trans. on Computers*, 60(6):753–766, 2011.

[10] S. Cho and H. Lee. Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance. In *IEEE/ACM Intl. Symp. on Microarchitecture (MICRO)*, 2009.

[11] B. Debnath, M. Mokbel, D. Lilja, and D. Du. Deferred updates for flash-based storage. In *IEEE Symp. on Mass Storage Systems and Technologies (MSST)*, 2010.

[12] P. Desnoyers. What systems researchers need to know about NAND flash. In *USENIX Hot Topics in Storage and File Systems (HotStorage)*, 2013.

[13] P. Desnoyers. Analytic models of SSD write performance. *Trans. Storage*, 10(2):8:1–8:25, Mar. 2014.

[14] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf. Characterizing flash memory: Anomalies, observations, and applications. In *IEEE/ACM Intl. Symp. on Microarchitecture (MICRO)*, 2009.

[15] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings. In *Intl. Conf. on Arch. Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[16] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam. Leveraging value locality in optimizing NAND flash-based SSDs. In *USENIX Conf. on File and Storage Technologies (FAST)*, 2011.

[17] C. Heegard. On the capacity of permanent memory. *IEEE Trans. on Inform. Theory*, 31(1):34–41, 1985.

[18] S. Huang, Q. Wei, J. Chen, C. Chen, and D. Feng. Improving flash-based disk cache with lazy adaptive replacement. In *IEEE Symp. on Mass Storage Systems and Technologies (MSST)*, 2013.

[19] Hynix. 32Gbit (4096M x 8bit) legacy NAND flash memory. Datasheet, 2012.

[20] S. Im and D. Shin. ComboFTL: Improving performance and lifespan of MLC flash memory using SLC flash buffer. *J. Syst. Archit.*, 56(12):641–653, Dec. 2010.

[21] A. N. Jacobvitz, R. Calderbank, and D. J. Sorin. Writing cosets of a convolutional code to increase the lifetime of flash memory. In *Allerton Conf. on Communication, Control, and Computing (Allerton)*, 2012.

[22] A. N. Jacobvitz, R. Calderbank, and D. J. Sorin. Coset coding to extend the lifetime of memory. In *IEEE Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2013.

[23] A. Jagmohan, M. Franceschini, and L. Lastras. Write amplification reduction in NAND flash through multi-write coding. In *IEEE Symp. on Mass Storage Systems and Technologies (MSST)*, 2010.

[24] J. Jeong, S. S. Hahn, S. Lee, and J. Kim. Lifetime improvement of NAND flash-based storage systems using dynamic program and erase scaling. In *USENIX Conf. on File and Storage Technologies (FAST)*, 2014.

[25] A. Jiang, Y. Li, E. Gad, M. Langberg, and J. Bruck. Joint rewriting and error correction in write-once memories. In *IEEE Intl. Symp. on Inform. Theory (ISIT)*, 2013.

[26] X. Jimenez, D. Novo, and P. Ienne. Libra: Software controlled cell bit-density to balance wear in NAND flash. *ACM Trans. Embed. Comput. Syst. (TECS)*, 14(2).

[27] X. Jimenez, D. Novo, and P. Ienne. Wear unleveling: Improving NAND flash lifetime by balancing page endurance. In *USENIX Conf. on File and Storage Technologies (FAST)*, 2014.

[28] T. Kgil, D. Roberts, and T. Mudge. Improving NAND flash based disk caches. In *Intl. Symp. on Computer Architecture (ISCA)*, 2008.

[29] H. Kim and S. Ahn. BPLRU: A buffer management scheme for improving random writes in flash storage. In *USENIX Conf. on File and Storage Technologies (FAST)*, 2008.

[30] S. Lee, T. Kim, K. Kim, and J. Kim. Lifetime management of flash-based SSDs using recovery-aware dynamic throttling. In *USENIX Conf. on File and Storage Technologies (FAST)*, 2012.

[31] C. Li, P. Shilane, F. Douglis, H. Shim, S. Smaldone, and G. Wallace. Nitro: A capacity-optimized SSD cache for primary storage. In *USENIX Annual Technical Conf. (ATC)*, 2014.

[32] J. Li and K. Mohanram. Write-once-memory-code phase change memory. In *Design, Automation and Test in Europe Conf. and Exhibition (DATE)*, 2014.

[33] R.-S. Liu, C.-L. Yang, and W. Wu. Optimizing NAND flash-based SSDs via retention relaxation. In *USENIX Conf. on File and Storage Technologies (FAST)*, 2012.

[34] Y. Lu, J. Shu, and W. Wang. ReconFS: A recon-

structable file system on flash storage. In *USENIX Conf. on File and Storage Technologies (FAST)*, 2014.

[35] X. Luojie, B. M. Kurkoski, and E. Yaakobi. WOM codes reduce write amplification in NAND flash memory. In *IEEE Global Communications Conf. (GLOBECOM)*, 2012.

[36] L. Marmol, S. Sundararaman, N. Talagala, R. Rangaswami, S. Devendrappa, B. Ramsundar, and S. Ganesan. NVMKV: A scalable and lightweight flash aware key-value store. In *USENIX Hot Topics in Storage and File Systems (HotStorage)*, 2014.

[37] N. Mielke, H. Belgal, I. Kalastirsky, P. Kalavade, A. Kurtz, Q. Meng, N. Righos, and J. Wu. Flash EEPROM threshold instabilities due to charge trapping during program/erase cycling. *IEEE Trans. on Device and Materials Reliability*, 4(3):335–344, Sept 2004.

[38] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom. SFS: Random write considered harmful in solid state drives. In *USENIX Conf. on File and Storage Technologies (FAST)*, 2012.

[39] V. Mohan, T. Bunker, L. Grupp, S. Gurumurthi, M. Stan, and S. Swanson. Modeling power consumption of NAND flash memories using FlashPower. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 32(7):1031–1044, July 2013.

[40] D. Narayanan, A. Donnelly, and A. Rowstron. Write off-loading: Practical power management for enterprise storage. *Trans. Storage*, 4(3):10:1–10:23, Nov. 2008.

[41] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron. Migrating server storage to SSDs: Analysis of tradeoffs. In *ACM European Conf. on Computer Systems (EuroSys)*, 2009.

[42] S. Odeh and Y. Cassuto. NAND flash architectures reducing write amplification through multi-write codes. In *IEEE Symp. on Mass Storage Systems and Technologies (MSST)*, 2014.

[43] Y. Oh, J. Choi, D. Lee, and S. H. Noh. Caching less for better performance: Balancing cache size and update cost of flash memory cache in hybrid storage systems. In *USENIX Conf. on File and Storage Technologies (FAST)*, 2012.

[44] Y. Pan, G. Dong, and T. Zhang. Exploiting memory device wear-out dynamics to improve NAND flash memory system performance. In *USENIX Conf. on File and Storage Technologies (FAST)*, 2011.

[45] K. Prall. Scaling non-volatile memory below 30nm. In *IEEE Non-Volatile Semiconductor Memory Workshop*, 2007.

[46] T. Pritchett and M. Thottethodi. SieveStore: A highly-selective, ensemble-level disk cache for cost-performance. In *Intl. Symp. on Computer Architecture (ISCA)*, 2010.

[47] R. L. Rivest and A. Shamir. How to Reuse a Write-Once Memory. *Inform. and Contr.*, 55(1-3):1–19, Dec. 1982.

[48] D. Roberts, T. Kgil, and T. Mudge. Integrating NAND flash devices onto servers. *Commun. ACM*, 52(4):98–103, Apr. 2009.

[49] Samsung. 16Gb F-die NAND flash, multi-level-cell (2bit/cell). Datasheet, 2011.

[50] G. Sarkis, P. Giard, A. Vardy, C. Thibeault, and W. J. Gross. Fast polar decoders: Algorithm and implementation. *IEEE Journal on Selected Areas in Communications*, 32(5):946–957, May 2014.

[51] A. Shpilka. New constructions of WOM codes using the Wozencraft Ensemble. *IEEE Trans. on Inform. Theory*, 59(7):4520–4529, 2013.

[52] K. Smith. Understanding SSD over-provisioning. *EDN Network*, January 2013.

[53] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber. Extending SSD lifetimes with disk-based write caches. In *USENIX Conf. on File and Storage Technologies (FAST)*, 2010.

[54] R. Stoica and A. Ailamaki. Improving flash write performance by using update frequency. *Proc. VLDB Endow.*, 6(9):733–744, July 2013.

[55] Toshiba. 32 GBIT (4G X 8 BIT) CMOS NAND E2PROM. Datasheet, 2011.

[56] E. Yaakobi, S. Kayser, P. H. Siegel, A. Vardy, and J. K. Wolf. Codes for write-once memories. *IEEE Trans. on Inform. Theory*, 58(9):5985–5999, 2012.

[57] E. Yaakobi, J. Ma, L. Grupp, P. H. Siegel, S. Swanson, and J. K. Wolf. Error characterization and coding schemes for flash memories. In *IEEE GLOBECOM Workshops (GC Wkshps)*, 2010.

[58] C. Zambelli, M. Indaco, M. Fabiano, S. Di Carlo, P. Prinetto, P. Olivo, and D. Bertozzi. A cross-layer approach for new reliability-performance trade-offs in MLC NAND flash memories. In *Design, Automation Test in Europe Conf. Exhibition (DATE)*, 2012.

[59] X. Zhang, L. Jang, Y. Zhang, C. Zhang, and J. Yang. WoM-SET: Low power proactive-SET-based PCM write using WoM code. In *IEEE Intl. Symp. on Low Power Electronics and Design (ISLPED)*, 2013.

[60] K. Zhao, W. Zhao, H. Sun, X. Zhang, N. Zheng, and T. Zhang. LDPC-in-SSD: Making advanced error correction codes work effectively in solid state drives. In *USENIX Conf. on File and Storage Technologies (FAST)*, 2013.

F2FS: A New File System for Flash Storage

Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho

S/W Development Team
Memory Business
Samsung Electronics Co., Ltd.

Abstract

F2FS is a Linux file system designed to perform well on modern flash storage devices. The file system builds on append-only logging and its key design decisions were made with the characteristics of flash storage in mind. This paper describes the main design ideas, data structures, algorithms and the resulting performance of F2FS.

Experimental results highlight the desirable performance of F2FS; on a state-of-the-art mobile system, it outperforms EXT4 under synthetic workloads by up to 3.1× (iozone) and 2× (SQLite). It reduces elapsed time of several realistic workloads by up to 40%. On a server system, F2FS is shown to perform better than EXT4 by up to 2.5× (SATA SSD) and 1.8× (PCIe SSD).

1 Introduction

NAND flash memory has been used widely in various mobile devices like smartphones, tablets and MP3 players. Furthermore, server systems started utilizing flash devices as their primary storage. Despite its broad use, flash memory has several limitations, like erase-before-write requirement, the need to write on erased blocks sequentially and limited write cycles per erase block.

In early days, many consumer electronic devices directly utilized "bare" NAND flash memory put on a platform. With the growth of storage needs, however, it is increasingly common to use a "solution" that has multiple flash chips connected through a dedicated controller. The firmware running on the controller, commonly called FTL (flash translation layer), addresses the NAND flash memory's limitations and provides a generic block device abstraction. Examples of such a flash storage solution include eMMC (embedded multimedia card), UFS (universal flash storage) and SSD (solid-state drive). Typically, these modern flash storage devices show much lower access latency than a hard disk drive (HDD), their mechanical counterpart. When it comes to random I/O, SSDs perform orders of magnitude better than HDDs.

However, under certain usage conditions of flash storage devices, the idiosyncrasy of the NAND flash media manifests. For example, Min et al. [21] observe that frequent random writes to an SSD would incur internal fragmentation of the underlying media and degrade the sustained SSD performance. Studies indicate that random write patterns are quite common and even more taxing to resource-constrained flash solutions on mobile devices. Kim et al. [12] quantified that the Facebook mobile application issues 150% and WebBench register 70% more random writes than sequential writes. Furthermore, over 80% of total I/Os are random and more than 70% of the random writes are triggered with `fsync` by applications such as Facebook and Twitter [8]. This specific I/O pattern comes from the dominant use of SQLite [2] in those applications. Unless handled carefully, frequent random writes and flush operations in modern workloads can seriously increase a flash device's I/O latency and reduce the device lifetime.

The detrimental effects of random writes could be reduced by the log-structured file system (LFS) approach [27] and/or the copy-on-write strategy. For example, one might anticipate file systems like BTRFS [26] and NILFS2 [15] would perform well on NAND flash SSDs; unfortunately, they do not consider the characteristics of flash storage devices and are inevitably suboptimal in terms of performance and device lifetime. We argue that traditional file system design strategies for HDDs—albeit beneficial—fall short of fully leveraging and optimizing the usage of the NAND flash media.

In this paper, we present the design and implementation of F2FS, a new file system optimized for modern flash storage devices. As far as we know, F2FS is the first publicly and widely available file system that is designed from scratch to optimize performance and lifetime of flash devices with a generic block interface.[1] This paper describes its design and implementation.

Listed in the following are the main considerations for the design of F2FS:

• **Flash-friendly on-disk layout (Section 2.1).** F2FS employs three configurable units: *segment*, *section* and *zone*. It allocates storage blocks in the unit of segments from a number of individual zones. It performs "cleaning" in the unit of section. These units are introduced to align with the underlying FTL's operational units to avoid unnecessary (yet costly) data copying.

• **Cost-effective index structure (Section 2.2).** LFS writes data and index blocks to newly allocated free space. If a leaf data block is updated (and written to somewhere), its direct index block should be updated, too. Once the direct index block is written, again its indirect index block should be updated. Such recursive updates result in a chain of writes, creating the "wandering tree" problem [4]. In order to attack this problem, we propose a novel index table called *node address table*.

• **Multi-head logging (Section 2.4).** We devise an effective hot/cold data separation scheme applied during logging time (i.e., block allocation time). It runs multiple active log segments concurrently and appends data and metadata to separate log segments based on their anticipated update frequency. Since the flash storage devices exploit media parallelism, multiple active segments can run simultaneously without frequent management operations, making performance degradation due to multiple logging (vs. single-segment logging) insignificant.

• **Adaptive logging (Section 2.6).** F2FS builds basically on append-only logging to turn random writes into sequential ones. At high storage utilization, however, it changes the logging strategy to threaded logging [23] to avoid long write latency. In essence, threaded logging writes new data to free space in a dirty segment without cleaning it in the foreground. This strategy works well on modern flash devices but may not do so on HDDs.

• `fsync` **acceleration with roll-forward recovery (Section 2.7).** F2FS optimizes small synchronous writes to reduce the latency of `fsync` requests, by minimizing required metadata writes and recovering synchronized data with an efficient roll-forward mechanism.

In a nutshell, F2FS builds on the concept of LFS but deviates significantly from the original LFS proposal with new design considerations. We have implemented F2FS as a Linux file system and compare it with two state-of-the-art Linux file systems—EXT4 and BTRFS. We also evaluate NILFS2, an alternative implementation of LFS in Linux. Our evaluation considers two generally categorized target systems: mobile system and server system. In the case of the server system, we study the file systems on a SATA SSD and a PCIe SSD. The results we obtain and present in this work highlight the overall desirable performance characteristics of F2FS.

In the remainder of this paper, Section 2 first describes the design and implementation of F2FS. Section 3 provides performance results and discussions. We describe related work in Section 4 and conclude in Section 5.

2 Design and Implementation of F2FS

2.1 On-Disk Layout

The on-disk data structures of F2FS are carefully laid out to match how underlying NAND flash memory is organized and managed. As illustrated in Figure 1, F2FS divides the whole volume into fixed-size *segments*. The segment is a basic unit of management in F2FS and is used to determine the initial file system metadata layout.

A *section* is comprised of consecutive segments, and a *zone* consists of a series of sections. These units are important during logging and cleaning, which are further discussed in Section 2.4 and 2.5.

F2FS splits the entire volume into six areas:

• **Superblock (SB)** has the basic partition information and default parameters of F2FS, which are given at the format time and not changeable.

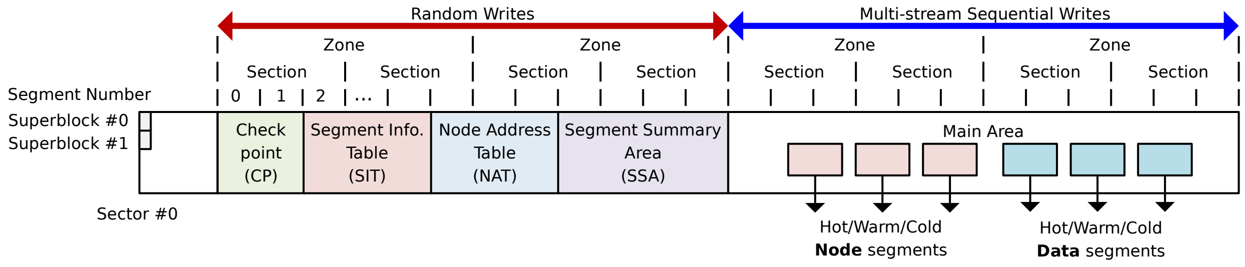
• **Checkpoint (CP)** keeps the file system status, bitmaps for valid NAT/SIT sets (see below), orphan inode lists and summary entries of currently active segments. A successful "checkpoint pack" should store a consistent F2FS status at a given point of time—a recovery point after a sudden power-off event (Section 2.7). The CP area stores two checkpoint packs across the two segments (#0 and #1): one for the last stable version and the other for the intermediate (obsolete) version, alternatively.

• **Segment Information Table (SIT)** contains per-segment information such as the number of valid blocks and the bitmap for the validity of all blocks in the "Main" area (see below). The SIT information is retrieved to select victim segments and identify valid blocks in them during the cleaning process (Section 2.5).

• **Node Address Table (NAT)** is a block address table to locate all the "node blocks" stored in the Main area.

• **Segment Summary Area (SSA)** stores summary entries representing the owner information of all blocks in the Main area, such as parent inode number and its

[1]F2FS has been available in the Linux kernel since version 3.8 and has been adopted in commercial products.



Random Writes
Multi-stream Sequential Writes
Zone
Section
Segment Number
0 1 2 ...
Superblock #0
Superblock #1
Check point (CP)
Segment Info. Table (SIT)
Node Address Table (NAT)
Segment Summary Area (SSA)
Main Area
Sector #0
Hot/Warm/Cold **Node** segments
Hot/Warm/Cold **Data** segments

Figure 1: On-disk layout of F2FS.

node/data offsets. The SSA entries identify parent node blocks before migrating valid blocks during cleaning.

• **Main Area** is filled with 4KB blocks. Each block is allocated and typed to be *node* or *data*. A node block contains inode or indices of data blocks, while a data block contains either directory or user file data. Note that a section does not store data and node blocks simultaneously.

Given the above on-disk data structures, let us illustrate how a file look-up operation is done. Assuming a file "`/dir/file`", F2FS performs the following steps: (1) It obtains the root inode by reading a block whose location is obtained from NAT; (2) In the root inode block, it searches for a directory entry named `dir` from its data blocks and obtains its inode number; (3) It translates the retrieved inode number to a physical location through NAT; (4) It obtains the inode named `dir` by reading the corresponding block; and (5) In the `dir` inode, it identifies the directory entry named `file`, and finally, obtains the file inode by repeating steps (3) and (4) for `file`. The actual data can be retrieved from the Main area, with indices obtained via the corresponding file structure.

2.2 File Structure

The original LFS introduced *inode map* to translate an inode number to an on-disk location. In comparison, F2FS utilizes the "node" structure that extends the inode map to locate more indexing blocks. Each node block has a unique identification number, "node ID". By using node ID as an index, NAT serves the physical locations of all node blocks. A node block represents one of three types: inode, direct and indirect node. An inode block contains a file's metadata, such as file name, inode number, file size, atime and dtime. A direct node block contains block addresses of data and an indirect node block has node IDs locating another node blocks.

As illustrated in Figure 2, F2FS uses pointer-based file indexing with direct and indirect node blocks to eliminate update propagation (i.e., "wandering tree" problem [27]). In the traditional LFS design, if a leaf data is updated, its direct and indirect pointer blocks are updated

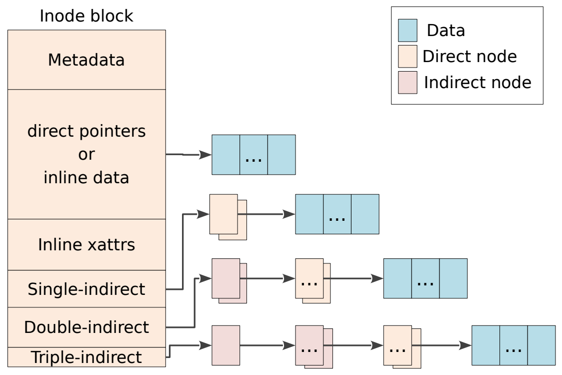


Figure 2: File structure of F2FS.

recursively. F2FS, however, only updates one direct node block and its NAT entry, effectively addressing the wandering tree problem. For example, when a 4KB data is appended to a file of 8MB to 4GB, the LFS updates two pointer blocks recursively while F2FS updates only one direct node block (not considering cache effects). For files larger than 4GB, the LFS updates one more pointer block (three total) while F2FS still updates only one.

An inode block contains direct pointers to the file's data blocks, two single-indirect pointers, two double-indirect pointers and one triple-indirect pointer. F2FS supports *inline data* and *inline extended attributes*, which embed small-sized data or extended attributes in the inode block itself. Inlining reduces space requirements and improve I/O performance. Note that many systems have small files and a small number of extended attributes. By default, F2FS activates inlining of data if a file size is smaller than 3,692 bytes. F2FS reserves 200 bytes in an inode block for storing extended attributes.

2.3 Directory Structure

In F2FS, a 4KB directory entry ("dentry") block is composed of a bitmap and two arrays of slots and names in pairs. The bitmap tells whether each slot is valid or not. A slot carries a hash value, inode number, length of a file name and file type (e.g., normal file, directory and symbolic link). A directory file constructs multi-level hash tables to manage a large number of dentries efficiently.

When F2FS looks up a given file name in a directory, it first calculates the hash value of the file name. Then, it traverses the constructed hash tables incrementally from level 0 to the maximum allocated level recorded in the inode. In each level, it scans one bucket of two or four dentry blocks, resulting in an *O(log(# of dentries))* complexity. To find a dentry more quickly, it compares the bitmap, the hash value and the file name in order.

When large directories are preferred (e.g., in a server environment), users can configure F2FS to initially allocate space for many dentries. With a larger hash table at low levels, F2FS reaches to a target dentry more quickly.

2.4 Multi-head Logging

Unlike the LFS that has one large log area, F2FS maintains six major log areas to maximize the effect of hot and cold data separation. F2FS statically defines three levels of temperature—hot, warm and cold—for node and data blocks, as summarized in Table 1.

Direct node blocks are considered hotter than indirect node blocks since they are updated much more frequently. Indirect node blocks contain node IDs and are written only when a dedicated node block is added or removed. Direct node blocks and data blocks for directories are considered hot, since they have obviously different write patterns compared to blocks for regular files. Data blocks satisfying one of the following three conditions are considered cold:

• **Data blocks moved by cleaning** (see Section 2.5). Since they have remained valid for an extended period of time, we expect they will remain so in the near future.
• **Data blocks labeled "cold" by the user**. F2FS supports an extended attribute operation to this end.
• **Multimedia file data**. They likely show write-once and read-only patterns. F2FS identifies them by matching a file's extension against registered file extensions.

By default, F2FS activates six logs open for writing. The user may adjust the number of write streams to two or four at mount time if doing so is believed to yield better results on a given storage device and platform. If six logs are used, each logging segment corresponds directly to a temperature level listed in Table 1. In the case of four logs, F2FS combines the cold and warm logs in each of node and data types. With only two logs, F2FS allocates one for node and the other for data types. Section 3.2.3 examines how the number of logging heads affects the effectiveness of data separation.

F2FS introduces configurable zones to be compatible with an FTL, with a view to mitigating the

Table 1: Separation of objects in multiple active segments.

Type	Temp.	Objects
Node	Hot	Direct node blocks for directories
	Warm	Direct node blocks for regular files
	Cold	Indirect node blocks
Data	Hot	Directory entry blocks
	Warm	Data blocks made by users
	Cold	Data blocks moved by cleaning; Cold data blocks specified by users; Multimedia file data

garbage collection (GC) overheads.[2] FTL algorithms are largely classified into three groups (block-associative, set-associative and fully-associative) according to the associativity between data and "log flash blocks" [24]. Once a data flash block is assigned to store initial data, log flash blocks assimilate data updates as much as possible, like the journal in EXT4 [18]. The log flash block can be used exclusively for a single data flash block (block-associative) [13], for all data flash blocks (fully-associative) [17], or for a set of contiguous data flash blocks (set-associative) [24]. Modern FTLs adopt a fully-associative or set-associative method, to be able to properly handle random writes. Note that F2FS writes node and data blocks in parallel using multi-head logging and an associative FTL would mix the separated blocks (in the file system level) into the same flash block. In order to avoid such misalignment, F2FS maps active logs to different zones to separate them in the FTL. This strategy is expected to be effective for set-associative FTLs. Multi-head logging is also a natural match with the recently proposed "multi-streaming" interface [10].

2.5 Cleaning

Cleaning is a process to reclaim scattered and invalidated blocks, and secures free segments for further logging. Because cleaning occurs constantly once the underlying storage capacity has been filled up, limiting the costs related with cleaning is extremely important for the sustained performance of F2FS (and any LFS in general). In F2FS, cleaning is done in the unit of a section.

F2FS performs cleaning in two distinct manners, *foreground* and *background*. Foreground cleaning is triggered only when there are not enough free sections, while a kernel thread wakes up periodically to conduct cleaning in background. A cleaning process takes three steps:

[2]Conducted by FTL, GC involves copying valid flash pages and erasing flash blocks for further data writes. GC overheads depend partly on how well file system operations align to the given FTL mapping algorithm.

(1) Victim selection. The cleaning process starts first to identify a victim section among non-empty sections. There are two well-known policies for victim selection during LFS cleaning—*greedy* and *cost-benefit* [11, 27]. The greedy policy selects a section with the smallest number of valid blocks. Intuitively, this policy controls overheads of migrating valid blocks. F2FS adopts the greedy policy for its foreground cleaning to minimize the latency visible to applications. Moreover, F2FS reserves a small unused capacity (5% of the storage space by default) so that the cleaning process has room for adequate operation at high storage utilization levels. Section 3.2.4 studies the impact of utilization levels on cleaning cost.

On the other hand, the cost-benefit policy is practiced in the background cleaning process of F2FS. This policy selects a victim section not only based on its utilization but also its "age". F2FS infers the age of a section by averaging the age of segments in the section, which, in turn, can be obtained from their last modification time recorded in SIT. With the cost-benefit policy, F2FS gets another chance to separate hot and cold data.

(2) Valid block identification and migration. After selecting a victim section, F2FS must identify valid blocks in the section quickly. To this end, F2FS maintains a validity bitmap per segment in SIT. Once having identified all valid blocks by scanning the bitmaps, F2FS retrieves parent node blocks containing their indices from the SSA information. If the blocks are valid, F2FS migrates them to other free logs.

For background cleaning, F2FS does not issue actual I/Os to migrate valid blocks. Instead, F2FS loads the blocks into page cache and marks them as dirty. Then, F2FS just leaves them in the page cache for the kernel worker thread to flush them to the storage later. This *lazy migration* not only alleviates the performance impact on foreground I/O activities, but also allows small writes to be combined. Background cleaning does not kick in when normal I/O or foreground cleaning is in progress.

(3) Post-cleaning process. After all valid blocks are migrated, a victim section is registered as a candidate to become a new free section (called a "pre-free" section in F2FS). After a checkpoint is made, the section finally becomes a free section, to be reallocated. We do this because if a pre-free section is reused before checkpointing, the file system may lose the data referenced by a previous checkpoint when unexpected power outage occurs.

2.6 Adaptive Logging

The original LFS introduced two logging policies, *normal logging* and *threaded logging*. In the normal logging, blocks are written to clean segments, yielding strictly sequential writes. Even if users submit many random write requests, this process transforms them to sequential writes as long as there exists enough free logging space. As the free space shrinks to nil, however, this policy starts to suffer high cleaning overheads, resulting in a serious performance drop (quantified to be over 90% under harsh conditions, see Section 3.2.5). On the other hand, threaded logging writes blocks to *holes* (invalidated, obsolete space) in existing dirty segments. This policy requires no cleaning operations, but triggers random writes and may degrade performance as a result.

F2FS implements both policies and switches between them dynamically according to the file system status. Specifically, if there are more than k clean sections, where k is a pre-defined threshold, normal logging is initiated. Otherwise, threaded logging is activated. k is set to 5% of total sections by default and can be configured.

There is a chance that threaded logging incurs undesirable random writes when there are scattered holes. Nevertheless, such random writes typically show better spatial locality than those in update-in-place file systems, since all holes in a dirty segment are filled first before F2FS searches for more in other dirty segments. Lee et al. [16] demonstrate that flash storage devices show better random write performance with strong spatial locality. F2FS gracefully gives up normal logging and turns to threaded logging for higher sustained performance, as will be shown in Section 3.2.5.

2.7 Checkpointing and Recovery

F2FS implements *checkpointing* to provide a consistent recovery point from a sudden power failure or system crash. Whenever it needs to remain a consistent state across events like `sync`, `umount` and foreground cleaning, F2FS triggers a checkpoint procedure as follows: (1) All dirty node and dentry blocks in the page cache are flushed; (2) It suspends ordinary writing activities including system calls such as `create`, `unlink` and `mkdir`; (3) The file system metadata, NAT, SIT and SSA, are written to their dedicated areas on the disk; and (4) Finally, F2FS writes a *checkpoint pack*, consisting of the following information, to the CP area:

• **Header and footer** are written at the beginning and the end of the pack, respectively. F2FS maintains in the header and footer a version number that is incremented on creating a checkpoint. The version number discriminates the latest stable pack between two recorded packs during the mount time;

• **NAT and SIT bitmaps** indicate the set of NAT and SIT blocks comprising the current pack;

• **NAT and SIT journals** contain a small number of recently modified entries of NAT and SIT to avoid frequent NAT and SIT updates;

• **Summary blocks of active segments** consist of in-memory SSA blocks that will be flushed to the SSA area in the future; and

• **Orphan blocks** keep "orphan inode" information. If an inode is deleted before it is closed (e.g., this can happen when two processes open a common file and one process deletes it), it should be registered as an orphan inode, so that F2FS can recover it after a sudden power-off.

2.7.1 Roll-Back Recovery

After a sudden power-off, F2FS rolls back to the latest consistent checkpoint. In order to keep at least one stable checkpoint pack while creating a new pack, F2FS maintains two checkpoint packs. If a checkpoint pack has identical contents in the header and footer, F2FS considers it valid. Otherwise, it is dropped.

Likewise, F2FS also manages two sets of NAT and SIT blocks, distinguished by the NAT and SIT bitmaps in each checkpoint pack. When it writes updated NAT or SIT blocks during checkpointing, F2FS writes them to one of the two sets alternatively, and then mark the bitmap to point to its new set.

If a small number of NAT or SIT entries are updated frequently, F2FS would write many 4KB-sized NAT or SIT blocks. To mitigate this overhead, F2FS implements a *NAT and SIT journal* within the checkpoint pack. This technique reduces the number of I/Os, and accordingly, the checkpointing latency as well.

During the recovery procedure at mount time, F2FS searches valid checkpoint packs by inspecting headers and footers. If both checkpoint packs are valid, F2FS picks the latest one by comparing their version numbers. Once selecting the latest valid checkpoint pack, it checks whether orphan inode blocks exist or not. If so, it truncates all the data blocks referenced by them and lastly frees the orphan inodes, too. Then, F2FS starts file system services with a consistent set of NAT and SIT blocks referenced by their bitmaps, after the roll-forward recovery procedure is done successfully, as is explained below.

2.7.2 Roll-Forward Recovery

Applications like database (e.g., SQLite) frequently write small data to a file and conduct `fsync` to guarantee durability. A naïve approach to supporting `fsync` would be to trigger checkpointing and recover data with the roll-back model. However, this approach leads to poor performance, as checkpointing involves writing all node and dentry blocks unrelated to the database file.

Table 2: Platforms used in experimentation. Numbers in parentheses are basic sequential and random performance *(Seq-R, Seq-W, Rand-R, Rand-W)* in MB/s.

Target	**System**	**Storage Devices**
Mobile	CPU: Exynos 5410 Memory: 2GB OS: Linux 3.4.5 Android: JB 4.2.2	eMMC 16GB: 2GB partition: *(114, 72, 12, 12)*
Server	CPU: Intel i7-3770 Memory: 4GB OS: Linux 3.14 Ubuntu 12.10 server	SATA SSD 250GB: *(486, 471, 40, 140)* PCIe (NVMe) SSD 960GB: *(1,295, 922, 41, 254)*

F2FS implements an efficient roll-forward recovery mechanism to enhance `fsync` performance. The key idea is to write data blocks and their direct node blocks only, excluding other node or F2FS metadata blocks. In order to find the data blocks selectively after rolling back to the stable checkpoint, F2FS remains a special flag inside direct node blocks.

F2FS performs roll-forward recovery as follows. If we denote the log position of the last stable checkpoint as **N**, (1) F2FS collects the direct node blocks having the special flag located in **N+n**, while constructing a list of their node information. **n** refers to the number of blocks updated since the last checkpoint. (2) By using the node information in the list, it loads the most recently written node blocks, named **N-n**, into the page cache. (3) Then, it compares the data indices in between **N-n** and **N+n**. (4) If it detects different data indices, then it refreshes the cached node blocks with the new indices stored in **N+n**, and finally marks them as dirty. Once completing the roll-forward recovery, F2FS performs checkpointing to store the whole in-memory changes to the disk.

3 Evaluation

3.1 Experimental Setup

We evaluate F2FS on two broadly categorized target systems, mobile system and server system. We employ a Galaxy S4 smartphone to represent the mobile system and an x86 platform for the server system. Specifications of the platforms are summarized in Table 2.

For the target systems, we back-ported F2FS from the 3.15-rc1 main-line kernel to the 3.4.5 and 3.14 kernel, respectively. In the mobile system, F2FS runs on a state-of-the-art eMMC storage. In the case of the server system, we harness a SATA SSD and a (higher-speed) PCIe

Table 3: Summary of benchmarks.

Target	Name	Workload	Files	File size	Threads	R/W	`fsync`
Mobile	iozone	Sequential and random read/write	1	1G	1	50/50	N
	SQLite	Random writes with frequent `fsync`	2	3.3MB	1	0/100	Y
	Facebook-app	Random writes with frequent `fsync`	579	852KB	1	1/99	Y
	Twitter-app	generated by the given system call traces	177	3.3MB	1	1/99	Y
Server	videoserver	Mostly sequential reads and writes	64	1GB	48	20/80	N
	fileserver	Many large files with random writes	80,000	128KB	50	70/30	N
	varmail	Many small files with frequent `fsync`	8,000	16KB	16	50/50	Y
	oltp	Large files with random writes and `fsync`	10	800MB	211	1/99	Y

SSD. Note that the values in the parentheses denoted under each storage device indicate the basic sequential read/write and random read/write bandwidth in MB/s. We measured the bandwidth through a simple single-thread application that triggers 512KB sequential I/Os and 4KB random I/Os with `O_DIRECT`.

We compare F2FS with EXT4 [18], BTRFS [26] and NILFS2 [15]. EXT4 is a widely used update-in-place file system. BTRFS is a copy-on-write file system, and NILFS2 is an LFS.

Table 3 summarizes our benchmarks and their characteristics in terms of generated I/O patterns, the number of touched files and their maximum size, the number of working threads, the ratio of reads and writes (R/W) and whether there are `fsync` system calls. For the mobile system, we execute and show the results of iozone [22], to study basic file I/O performance. Because mobile systems are subject to costly random writes with frequent `fsync` calls, we run *mobibench* [8], a macro benchmark, to measure the SQLite performance. We also replay two system call traces collected from the "Facebook" and "Twitter" application (each dubbed "Facebook-app" and "Twitter-app") under a realistic usage scenario [8].

For the server workloads, we make use of a synthetic benchmark called Filebench [20]. It emulates various file system workloads and allows for fast intuitive system performance evaluation. We use four pre-defined workloads in the benchmark—videoserver, fileserver, varmail and oltp. They differ in I/O pattern and `fsync` usage.

Videoserver issues mostly sequential reads and writes. Fileserver pre-allocates 80,000 files with 128KB data and subsequently starts 50 threads, each of which creates and deletes files randomly as well as reads and appends small data to randomly chosen files. This workload, thus, represents a scenario having many large files touched by buffered random writes and no `fsync`. Varmail creates and deletes a number of small files with `fsync`, while oltp pre-allocates ten large files and updates their data randomly with `fsync` with 200 threads in parallel.

3.2 Results

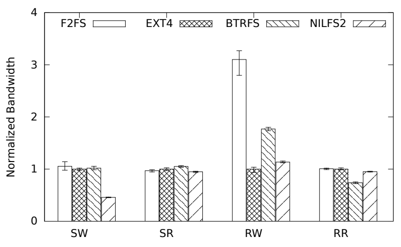
This section gives the performance results and insights obtained from deep block trace level analysis. We examined various I/O patterns (i.e., read, write, `fsync` and discard[3]), amount of I/Os and request size distribution. For intuitive and consistent comparison, we normalize performance results against EXT4 performance. We note that performance depends basically on the speed gap between sequential and random I/Os. In the case of the mobile system that has low computing power and a slow storage, I/O pattern and its quantity are the major performance factors. For the server system, CPU efficiency with instruction execution overheads and lock contention become an additional critical factor.

3.2.1 Performance on the Mobile System

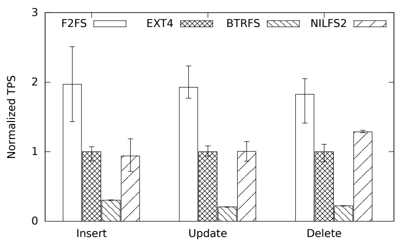
Figure 3(a) shows the iozone results of sequential read/write (SR/SW) and random read/write (RR/RW) bandwidth on a single 1GB file. In the SW case, NILFS2 shows performance degradation of nearly 50% over EXT4 since it triggers expensive synchronous writes periodically, according to its own data flush policy. In the RW case, F2FS performs 3.1× better than EXT4, since it turns over 90% of 4KB random writes into 512KB sequential writes (not directly shown in the plot). BTRFS also performs well (1.8×) as it produces sequential writes through the copy-on-write policy. While NILFS2 transforms random writes to sequential writes, it gains only 10% improvement due to costly synchronous writes. Furthermore, it issues up to 30% more write requests than other file systems. For RR, all file systems show comparable performance. BTRFS shows slightly lower performance due to its tree indexing overheads.

Figure 3(b) gives SQLite performance measured in transactions per second (TPS), normalized against that of EXT4. We measure three types of transactions—

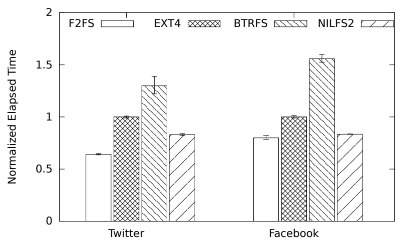
[3]A discard command gives a hint to the underlying flash storage device that a specified address range has no valid data. This command is sometimes called "trim" or "unmap".



(a) iozone

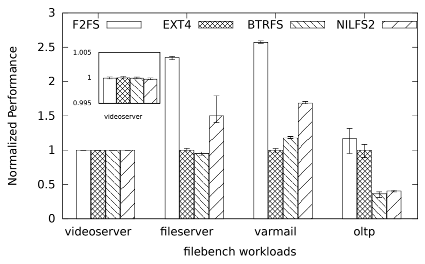


(b) SQLite

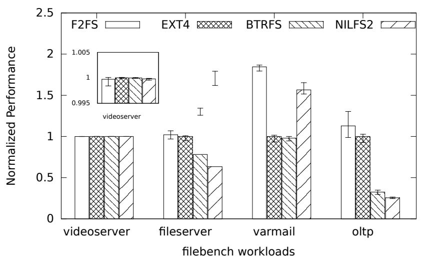


(c) Facebook-app and Twitter-app

Figure 3: Performance results on the mobile system.



(a) SATA SSD



(b) PCIe SSD

Figure 4: Performance results on the server system.

insert, update and delete—on a DB comprised of 1,000 records under the write ahead logging (WAL) journal mode. This journal mode is considered the fastest in SQLite. F2FS shows significantly better performance than other file systems and outperforms EXT4 by up to 2×. For this workload, the roll-forward recovery policy of F2FS produces huge benefits. In fact, F2FS reduces the amount of data writes by about 46% over EXT4 in all examined cases. Due to heavy indexing overheads, BTRFS writes 3× more data than EXT4, resulting in performance degradation of nearly 80%. NILFS2 achieves similar performance with a nearly identical amount of data writes compared to EXT4.

Figure 3(c) shows normalized elapsed times to complete replaying the Facebook-app and Twitter-app traces. They resort to SQLite for storing data, and F2FS reduces the elapsed time by 20% (Facebook-app) and 40% (Twitter-app) compared to EXT4.

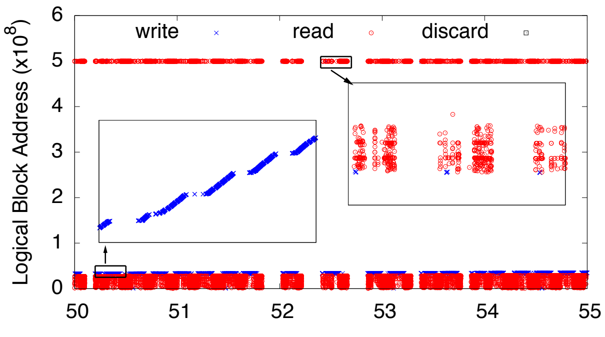
3.2.2 Performance on the Server System

Figure 4 plots performance of the studied file systems using SATA and PCIe SSDs. Each bar indicates normalized performance (i.e., performance improvement if the bar has a value larger than 1).

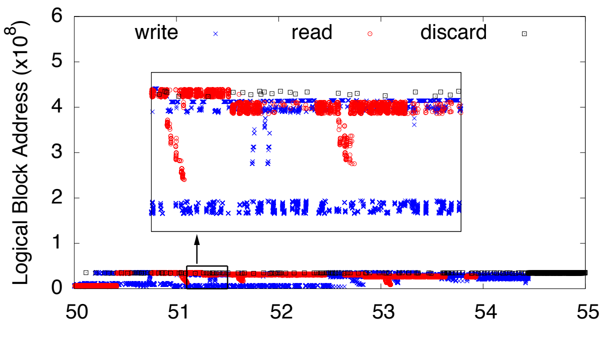
Videoserver generates mostly sequential reads and writes, and all results, regardless of the device used, expose no performance gaps among the studied file systems. This demonstrates that F2FS has no performance regression for normal sequential I/Os.

Fileserver has different I/O patterns; Figure 5 compares block traces obtained from all file systems on the SATA SSD. A closer examination finds that only 0.9% of all write requests generated by EXT4 are for 512KB, while F2FS has 6.9% (not directly shown in the plot). Another finding is that EXT4 issues many small discard commands and causes visible command processing overheads, especially on the SATA drive; it trims two thirds of all block addresses covered by data writes and nearly 60% of all discard commands were for an address space smaller than 256KB in size. In contrast, F2FS discards obsolete spaces in the unit of segments only when checkpointing is triggered; it trims 38% of block address space with no small discard commands. These differences lead to a 2.4× performance gain (Figure 4(a)).

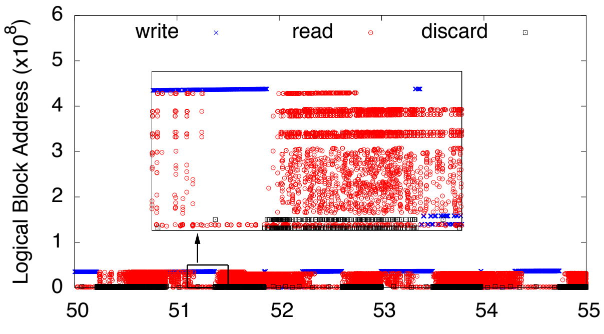
On the other hand, BTRFS degrades performance by 8%, since it issues 512KB data writes in only 3.8% of all write requests. In addition, it trims 47% of block address space with small discard commands (corresponding to 75% of all discard commands) during the read service time as shown in Figure 5(c). In the case of NILFS2, as many as 78% of its write requests are for 512KB (Figure 5(d)). However, its periodic synchronous data flushes limited the performance gain over EXT4 to 1.8×. On the PCIe SSD, all file systems perform rather similarly. This is because the PCIe SSD used in the study performs concurrent buffered writes well.



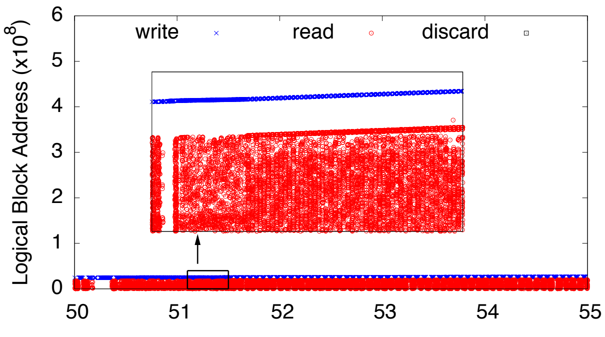
(a) F2FS



(b) EXT4



(c) BTRFS



(d) NILFS2

Figure 5: Block traces of the fileserver workload according to the running time in seconds.

In the varmail case, F2FS outperforms EXT4 by 2.5× on the SATA SSD and 1.8× on the PCIe SSD, respectively. Since varmail generates many small writes with concurrent `fsync`, the result again underscores the efficiency of `fsync` processing in F2FS. BTRFS performance was on par with that of EXT4 and NILFS2 performed relatively well on the PCIe SSD.

The oltp workload generates a large number of random writes and `fsync` calls on a single 800MB database file (unlike varmail, which touches many small files). F2FS shows measurable performance advantages over EXT4—16% on the SATA SSD and 13% on the PCIe SSD. On the other hand, both BTRFS and NILFS2 performed rather poorly on the PCIe drive. Fast command processing and efficient random writes on the PCIe drive appear to move performance bottleneck points, and BTRFS and NILFS2 do not show robust performance.

Our results so far have clearly demonstrated the relative effectiveness of the overall design and implementation of F2FS. We will now examine the impact of F2FS logging and cleaning policies.

3.2.3 Multi-head Logging Effect

This section studies the effectiveness of the multi-head logging policy of F2FS. Rather than presenting extensive evaluation results that span many different workloads, we focus on an experiment that captures the intuitions of our design. The metric used in this section is the *number of valid blocks* in a given dirty segment before cleaning. If hot and cold data separation is done perfectly, a dirty segment would have either zero valid blocks or the maximum number of valid blocks in a segment (512 under the default configuration). An aged dirty segment would carry zero valid blocks in it if all (hot) data stored in the segment have been invalidated. By comparison, a dirty segment full of valid blocks is likely keeping cold data.

In our experiment, we run two workloads simultaneously: varmail and copying of jpeg files. Varmail employs 10,000 files in total in 100 directories and writes 6.5GB of data. We copy 5,000 jpeg files of roughly 500KB each, hence resulting in 2.5GB of data written. Note that F2FS statically classifies jpeg files as cold data. After these workloads finish, we count the number of valid blocks in all dirty segments. We repeat the experiment as we vary the number of logs from two to six.

Figure 6 gives the result. With two logs, over 75% of all segments have more than 256 valid blocks while "full segments" with 512 valid blocks are very few. Because the two-log configuration splits only data segments (85% of all dirty segments, not shown) and node segments (15%), the effectiveness of multi-head logging is fairly limited. Adding two more logs changes the picture

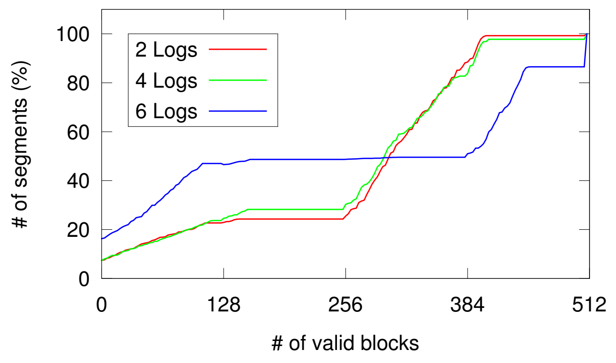


Figure 6: Dirty segment distribution according to the number of valid blocks in segments.

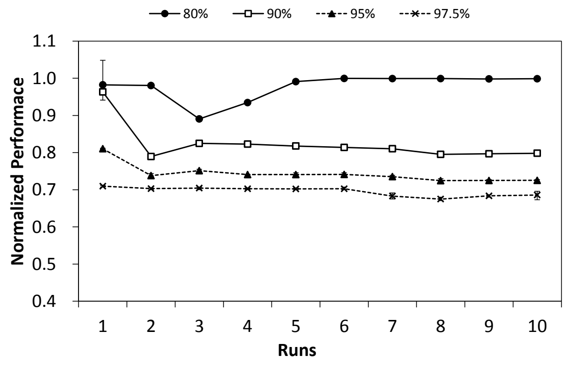
somewhat; it increases the number of segments having fewer than 256 valid blocks. It also slightly increases the number of nearly full segments.

Lastly, with six logs, we clearly see the benefits of hot and cold data separation; the number of pre-free segments having zero valid blocks and the number of full segments increase significantly. Moreover, there are more segments having relatively few valid blocks (128 or fewer) and segments with many valid blocks (384 or more). An obvious impact of this bimodal distribution is improved cleaning efficiency (as cleaning costs depend on the number of valid blocks in a victim segment).

We make several observations before we close this section. First, the result shows that more logs, allowing finer separation of data temperature, generally bring more benefits. However, in the particular experiment we performed, the benefit of four logs over two logs was rather insignificant. If we separate cold data from hot and warm data (as defined in Table 1) rather than hot data from warm and cold data (default), the result would look different. Second, since the number of valid blocks in dirty segments will gradually decrease over time, the left-most knee of the curves in Figure 6 will move upward (at a different speed according to the chosen logging configuration). Hence, if we age the file system, we expect that multi-head logging benefits will become more visible. Fully studying these observations is beyond the scope of this paper.

3.2.4 Cleaning Cost

We quantify the impact of cleaning in F2FS in this section. In order to focus on file system level cleaning cost, we ensure that SSD level GC does not occur during experiments by intentionally leaving ample free space in the SSD. To do so, we format a 250GB SSD and obtain a partition of (only) 120GB.



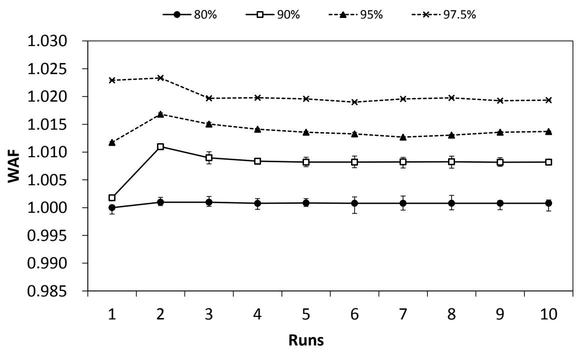
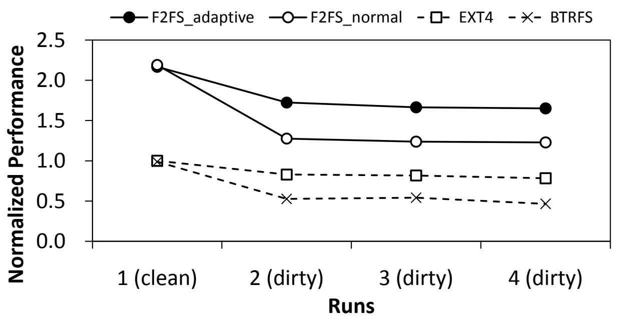


Figure 7: Relative performance (upper) and write amplification factor (lower) of the first ten runs. Four lines capture results for different file system utilization levels.

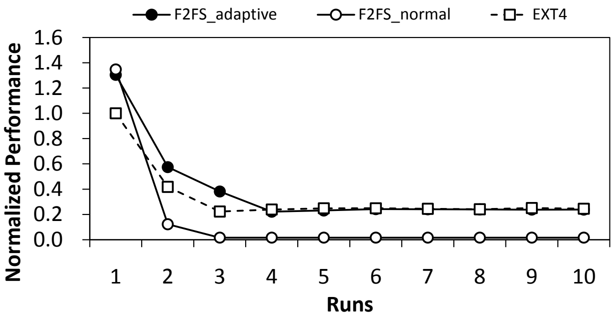
After reserving 5% of the space for overprovisioning (Section 2.5), we divide remaining capacity into "cold" and "hot" regions. We build four configurations that reflect different file system utilization levels by filling up the two regions as follows: 80% (60 (cold):20 (hot)), 90% (60:30), 95% (60:35) and 97.5% (60:37.5). Then, we iterate ten runs of experiments where each run randomly writes 20GB of data in 4KB to the hot region.

Figure 7 plots results of the first ten runs in two metrics: performance (throughput) and write amplification factor (WAF).[4] They are relative to results obtained on a clean SSD. We make two main observations. First, higher file system utilization leads to larger WAF and reduced performance. At 80%, performance degradation and WAF increase were rather minor. On the third run, the file system ran out of free segments and there was a performance dip. During this run, it switched to threaded logging from normal logging, and as the result, performance stabilized. (We revisit the effects of adaptive, threaded logging in Section 3.2.5.) After the third run, nearly all data were written via threaded logging, in place. In this case, cleaning is needed not for data, but

[4]Iterating 100 runs would not reveal further performance drops.



(a) fileserver (random operations) on a device filled up 94%.



(b) iozone (random updates) on a device filled up 100%.

Figure 8: Worst-case performance drop ratio under file system aging.

for recording nodes. As we raised utilization level from 80% to 97.5%, the amount of GC increased and the performance degradation became more visible. At 97.5%, the performance loss was about 30% and WAF 1.02.

The second observation is that F2FS does not dramatically increase WAF at high utilization levels; adaptive logging plays an important role of keeping WAF down. Note that threaded logging incurs random writes whereas normal logging issues sequential writes. While random writes are relatively expensive and motivates append-only logging as a preferred mode of operation in many file systems, our design choice (of switching to threaded logging) is justified because: cleaning could render very costly due to a high WAF when the file system is fragmented, and SSDs have high random write performance. Results in this section show that F2FS successfully controls the cost of cleaning at high utilization levels.

Showing the positive impact of background cleaning is not straightforward because background cleaning is suppressed during busy periods. Still, We measured over 10% performance improvement at a 90% utilization level when we insert an idle time of ten minutes or more between runs.

3.2.5 Adaptive Logging Performance

This section delves into the question: *How effective is the F2FS adaptive logging policy with threaded logging?* By default, F2FS switches to threaded logging from normal logging when the number of free sections falls below 5% of total sections. We compare this default configuration ("F2FS_adaptive") with "F2FS_normal", which sticks to the normal logging policy all the time. For experiments, we design and perform the following two intuitive tests on the SATA SSD.

• **fileserver test.** This test first fills up the target storage partition 94%, with hundreds of 1GB files. The test then runs the fileserver workload four times and measures the performance trends (Figure 8(a)). As we repeat experiments, the underlying flash storage device as well as the file system get fragmented. Accordingly, the performance of the workload is supposed to drop. Note that we were unable to perform this test with NILFS2 as it stopped with a "no space" error report.

EXT4 showed the mildest performance hit—17% between the first and the second round. By comparison, BTRFS and F2FS (especially F2FS_normal) saw a severe performance drop of 22% and 48% each, as they do not find enough sequential space. On the other hand, F2FS_adaptive serves 51% of total writes with threaded logging (not shown in the plot) and successfully limited performance degradation in the second round to 22% (comparable to BTRFS and not too far from EXT4). As the result, F2FS maintained the performance improvement ratio of two or more over EXT4 across the board. All the file systems were shown to sustain performance beyond the second round.

Further examination reveals that F2FS_normal writes 27% more data than F2FS_adaptive due to foreground cleaning. The large performance hit on BTRFS is due partly to the heavy usage of small discard commands.

• **iozone test.** This test first creates sixteen 4GB files and additional 1GB files until it fills up the device capacity (∼100%). Then it runs iozone to perform 4KB random writes on the sixteen 4GB files. The aggregate write volume amounts to 512MB per file. We repeat this step ten times, which turns out to be quite harsh, as both BTRFS and NILFS2 failed to complete with a "no space" error. Note that from the theoretical viewpoint, EXT4, an update-in-place file system, would perform the best in this test because EXT4 issues random writes without creating additional file system metadata. On the other hand, a log-structured file system like F2FS may suffer high cleaning costs. Also note that this workload fragments the data in the storage device, and the storage performance would suffer as the workload triggers repeated device-internal GC operations.

Under EXT4, the performance degradation was about 75% (Figure 8(b)). In the case of F2FS_normal, as expected, the performance drops to a very low level (of less than 5% of EXT4 from round 3) as both the file system and the storage device keep busy cleaning fragmented capacity to reclaim new space for logging. F2FS_adaptive is shown to handle the situation much more gracefully; it performs better than EXT4 in the first few rounds (when fragmentation was not severe) and shows performance very similar to that of EXT4 as the experiment advances with more random writes.

The two experiments in this section reveal that adaptive logging is critical for F2FS to sustain its performance at high storage utilization levels. The adaptive logging policy is also shown to effectively limit the performance degradation of F2FS due to fragmentation.

4 Related Work

This section discusses prior work related to ours in three categories—log-structured file systems, file systems targeting flash memory, and optimizations specific to FTL.

4.1 Log-Structured File Systems (LFS)

Much work has been done on log-structured file systems (for HDDs), beginning with the original LFS proposal by Rosenblum et al. [27]. Wilkes et al. proposed a hole plugging method in which valid blocks of a victim segment are moved to *holes*, i.e., invalid blocks in other dirty segment [30]. Matthews et al. proposed an adaptive cleaning policy where they choose between a normal logging policy and a hole-plugging policy based on cost-benefit evaluation [19]. Oh et al. [23] demonstrated that threaded logging provides better performance in a highly utilized volume. F2FS has been tuned on the basis of prior work and real-world workloads and devices.

A number of studies focus on separating hot and cold data. Wang and Hu [28] proposed to distinguish active and inactive data in the buffer cache, instead of writing them to a single log and separating them during cleaning. They determine which data is active by monitoring access patterns. Hylog [29] adopts a hybrid approach; it uses logging for hot pages to achieve high random write performance, and overwriting for cold pages to reduce cleaning cost.

SFS [21] is a file system for SSDs implemented based on NILFS2. Like F2FS, SFS uses logging to eliminate random writes. To reduce the cost of cleaning, they separate hot and cold data in the buffer cache, like [28], based on the "update likelihood" (or hotness) measured by tracking write counts and age per block. They use iterative quantization to partition segments into groups based on measured hotness.

Unlike the hot/cold data separation methods that resort to run-time monitoring of access patterns [21, 28], F2FS estimates update likelihood using information readily available, such as file operation (append or overwrite), file type (directory or regular file) and file extensions. While our experimental results show that the simple approach we take is fairly effective, more sophisticated run-time monitoring approaches can be incorporated in F2FS to fine-track data temperature.

NVMFS is an experimental file system assuming two distinct storage media: NVRAM and NAND flash SSD [25]. The fast byte-addressable storage capacity from NVRAM is used to store hot and meta data. Moreover, writes to the SSD are sequentialized as in F2FS.

4.2 Flash Memory File Systems

A number of file systems have been proposed and implemented for embedded systems that use raw NAND flash memories as storage [1, 3, 6, 14, 31]. These file systems directly access NAND flash memories while addressing all the chip-level issues such as wear-leveling and bad block management. Unlike these systems, F2FS targets flash storage devices that come with a dedicated controller and firmware (FTL) to handle low-level tasks. Such flash storage devices are more commonplace.

Josephson et al. proposed the direct file system (DFS) [9], which leverages special support from host-run FTL, including atomic update interface and very large logical address space, to simplify the file system design. DFS is however limited to specific flash devices and system configurations and is not open source.

4.3 FTL Optimizations

There has been much work aiming at improving random write performance at the FTL level, sharing some design strategies with F2FS. Most FTLs use a log-structured update approach to overcome the no-overwrite limitation of flash memory. DAC [5] provides a page-mapping FTL that clusters data based on update frequency by monitoring accesses at run time. To reduce the overheads of large page mapping tables, DFTL [7] dynamically loads a portion of the page map into working memory on demand and offers the random-write benefits of page mapping for devices with limited RAM.

Hybrid mapping (or log block mapping) is an extension of block mapping to improve random writes [13, 17, 24]. It has a smaller mapping table than page mapping while its performance can be as good as page mapping for workloads with substantial access locality.

5 Concluding Remarks

F2FS is a full-fledged Linux file system designed for modern flash storage devices and is slated for wider adoption in the industry. This paper describes key design and implementation details of F2FS. Our evaluation results underscore how our design decisions and trade-offs lead to performance advantages, over other existing file systems. F2FS is fairly young—it was incorporated in Linux kernel 3.8 in late 2012. We expect new optimizations and features will be continuously added to the file system.

Acknowledgment

Authors appreciate constructive comments of the reviewers and our shepherd, Ted Ts'o, which helped improve the quality of this paper. This work is a result of long and dedicated team efforts; without numerous contributions of prior F2FS team members (especially Jaegeuk Kim, Chul Lee, ByoungGeun Kim, Sehwan Lee, Seokyoung Ko, Dongbin Park and Sunghoon Park), this work would not have been possible.

References

[1] Unsorted block image file system. `http://www.linux-mtd.infradead.org/doc/ubifs.html`.

[2] Using databases in android: SQLite. `http://developer.android.com/guide/topics/data/data-storage.html#db`.

[3] Yet another flash file system. `http://www.yaffs.net/`.

[4] A. B. Bityutskiy. JFFS3 design issues. `http://www.linux-mtd.infradead.org`, 2005.

[5] M.-L. Chiang, P. C. Lee, and R.-C. Chang. Using data clustering to improve cleaning performance for flash memory. *Software-Practice and Experience*, 29(3):267–290, 1999.

[6] J. Engel and R. Mertens. LogFS-finally a scalable flash file system. In *Proceedings of the International Linux System Technology Conference*, 2005.

[7] A. Gupta, Y. Kim, and B. Urgaonkar. *DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings*, volume 44. ACM, 2009.

[8] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won. I/O stack optimization for smartphones. In *Proceedings of the USENIX Anual Technical Conference (ATC)*, pages 309–320, 2013.

[9] W. K. Josephson, L. A. Bongo, K. Li, and D. Flynn. DFS: A file system for virtualized flash storage. *ACM Transactions on Storage (TOS)*, 6(3):14:1–14:25, 2010.

[10] J.-U. Kang, J. Hyun, H. Maeng, and S. Cho. The multi-streamed solid-state drive. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, Philadelphia, PA, 2014. USENIX Association.

[11] A. Kawaguchi, S. Nishioka, and H. Motoda. A flash-memory based file system. In *Proceedings of the USENIX Anual Technical Conference (ATC)*, pages 155–164, 1995.

[12] H. Kim, N. Agrawal, and C. Ungureanu. Revisiting storage for smartphones. *ACM Transactions on Storage (TOS)*, 8(4):14, 2012.

[13] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho. A space-efficient flash translation layer for compactflash systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, 2002.

[14] J. Kim, H. Shim, S.-Y. Park, S. Maeng, and J.-S. Kim. Flashlight: A lightweight flash file system for embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 11(1):18, 2012.

[15] R. Konishi, Y. Amagai, K. Sato, H. Hifumi, S. Kihara, and S. Moriai. The Linux implementation of a log-structured file system. *ACM SIGOPS Operating Systems Review*, 40(3):102–107, 2006.

[16] S. Lee, D. Shin, Y.-J. Kim, and J. Kim. LAST: locality-aware sector translation for NAND flash memory-based storage systems. *ACM SIGOPS Operating Systems Review*, 42(6):36–42, 2008.

[17] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(3):18, 2007.

[18] 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. Citeseer, 2007.

[19] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson. Improving the performance of log-structured file systems with adaptive methods. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*,

pages 238–251, 1997.

[20] R. McDougall, J. Crase, and S. Debnath. Filebench: File system microbenchmarks. `http://www.opensolaris.org`, 2006.

[21] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom. SFS: Random write considered harmful in solid state drives. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 139–154, 2012.

[22] W. D. Norcott and D. Capps. Iozone filesystem benchmark. *URL: www.iozone.org*, 55, 2003.

[23] Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh. Optimizations of LFS with slack space recycling and lazy indirect block update. In *Proceedings of the Annual Haifa Experimental Systems Conference*, page 2, 2010.

[24] C. Park, W. Cheon, J. Kang, K. Roh, W. Cho, and J.-S. Kim. A reconfigurable FTL (flash translation layer) architecture for NAND flash-based applications. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(4):38, 2008.

[25] S. Qiu and A. L. N. Reddy. NVMFS: A hybrid file system for improving random write in nand-flash SSD. In *IEEE 29th Symposium on Mass Storage Systems and Technologies, MSST 2013, May 6-10, 2013, Long Beach, CA, USA*, pages 1–5, 2013.

[26] O. Rodeh, J. Bacik, and C. Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):9, 2013.

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

[28] J. Wang and Y. Hu. WOLF: A novel reordering write buffer to boost the performance of log-structured file systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 47–60, 2002.

[29] W. Wang, Y. Zhao, and R. Bunt. Hylog: A high performance approach to managing disk layout. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 144–158, 2004.

[30] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems (TOCS)*, 14(1):108–136, 1996.

[31] D. Woodhouse. JFFS: The journaling flash file system. In *Proceedings of the Ottawa Linux Symposium*, 2001.

A Practical Implementation of Clustered Fault Tolerant Write Acceleration in a Virtualized Environment

Deepavali Bhagwat Mahesh Patil Michal Ostrowski Murali Vilayannur
Woon Jung Chethan Kumar

PernixData, Inc.

Abstract

Host-side flash storage opens up an exciting avenue for accelerating Virtual Machine (VM) writes in virtualized datacenters. The key challenge with implementing such an acceleration layer is to do so without breaking live VM migration which is essential for providing distributed resource management and high availability. High availability also powers-on VMs on new host when the previous host crashes. We introduce FVP, a fault tolerant host-side flash write acceleration layer that seamlessly integrates with the virtualized environment while preserving dynamic resource management and high availability, the holy tenets of a virtualized environment. FVP integrates with the VMware ESX hypervisor kernel to intercept VM I/O and redirects the I/O to host-side flash devices. VMs experience flash latencies instead of SAN latencies and write intensive applications such as databases and email servers benefit from predictable write throughput. No changes are required to the VM guest operating systems so VM applications can continue to function seamlessly without any modifications. FVP pools together all the host-side flash devices in the cluster so every host can access another host's flash device preserving VM mobility. By replicating VM writes onto peer host-side flash devices, FVP is able to tolerate multiple cascading host and flash failures. Failure recovery is distributed, requiring no central co-ordination. We describe the workings of the FVP key components and demonstrate how FVP reduces VM latencies to accelerate VM writes, improves performance predictability, and increases virtualized datacenter efficiency.

1 Introduction

Virtualization has revolutionized how we build and operate data centers today. Large cost savings and increased operational efficiencies have made virtualization the biggest trend in data centers. However, as more applications become virtualized, and Virtual Machine (VM) density increases, shared storage performance does not scale with the high volumes of cumulative I/O generated by the VMs. I/O bottlenecks in the storage array add significant latency to virtual applications, resulting in slow response times at best and unusable applications at worst [1–3].

Provisioning better or more storage hardware is one way to address this problem. Some of these strategies include provisioning faster disks, improving the Storage Area Network (SAN) interconnect speeds, deploying flash caches in storage controllers [4–8], and replacing spinning disks with an all flash array [9, 10]. Replacing spinning disks with an all flash array is disruptive, incurring system downtime which may not be viable. Also, upgrading the SAN is usually a temporary fix in that even the new storage array will reach peak performance at some point resulting in the need for constant upgrades. Adding more CPUs or hosts is less disruptive.

An alternative approach is to install a flash device on the host and use it to cache VM writes. By co-locating the application's working set close to the application at the beginning of the I/O path, applications experience response times of the order of microseconds as opposed to milliseconds for shared storage. Host-side flash is thus used to *accelerate* applications and decouple storage performance from storage capacity [11–16].

Host-side flash has been typically used to cache recently accessed data to accelerate reads alone. VM reads are first issued to the flash device and, in case of a cache miss, issued to the SAN. The newly read data are also cached. VM writes are issued to the flash device and to the SAN. This approach of using the host-side flash device to accelerate reads is called *write-through* (*wt*) acceleration [12, 14]. Because a significant number of reads are offloaded from the SAN, it frees up the SAN's resources to service writes and un-cached reads. Therefore, write-through yields some improvement in write performance as well.

Another approach is to use the host-side flash to accelerate both writes *and* reads. This is called write-back (*wb*) acceleration [11]. In *wb*, VM writes are issued to flash and on flash write completion, acknowledged back to the VM without issuing them to the SAN. This accelerates the writes since writes complete at flash speed, not SAN speed. In the background, writes on the flash device are batched and then issued periodically to the SAN to flush dirty VM data and to free up flash space for future writes. This process of issuing batched writes to the SAN is called *destaging*.

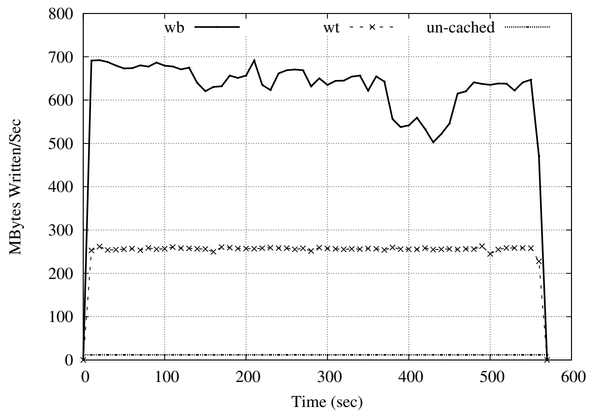


Figure 1: Combined throughput of Microsoft Exchange Server JetStress and fio, in *wb*, *wt*, un-cached

Write-back acceleration has a significant impact on write performance, because writes are acknowledged to the VM as soon as they are committed to the flash device alone. This creates a very short I/O path, resulting in very low latencies (typically microseconds). Write-through, in contrast, requires writes to cross the storage fabric to get to the SAN and be acknowledged before completion. Hence, VM writes in *wt* incurs SAN latencies. Figure 1 depicts the combined throughput of two VMs, one running Microsoft Exchange Server Jetstress [17] and the other running fio [18]. JetStress simulates the workload of an Exchange database consisting of transactions (reads and writes) by issuing 32 KB random reads and writes, and 14 KB sequential writes. fio was setup to issue 64 K sequential writes. Clearly, *wb* yields much better throughput than *wt*. Though the theory is sound and these benefits are appealing, accelerating writes using host-side flash in a clustered virtualized environment is not trivial.

The first challenge using host side flash in virtualized environments is that it must be done without breaking VM mobility [19]. For Distributed Resource Scheduling and Power Management (DRS) [20, 21], to balance resources utilization, VMs are live migrated or re-distributed across hosts. For High Availability (HA) [22], in case of a host failure, VMs are migrated away from the failed host. VM mobility, therefore, must be preserved.

To ensure VM mobility, *all* the hosts *must* have access to the VM's data. Virtualized datacenters achieve this by consolidating storage arrays under a shared SAN. Host-side flash, in contrast, is a local resource so hosts cannot access each others flash. If a VM with data on a host's flash device gets migrated to another host, the VM loses access to its data. This precludes VM mobility and consequently breaks DRS and HA. Byan *et al.* [12] and Holland *et al.* [13] cite these reasons for why host-side flash cannot be used for accelerating writes.

The second challenge using host-side flash for accelerating writes is that it creates potential fault tolerance and consistency problems. In case of a flash device failure VM writes which were not yet destaged are not retrievable resulting in data loss. Koller *et al.* [11] argue that the only way to *prevent* data loss is to retrench to *wt*.

The third challenge using host-side flash for accelerating writes is that write-heavy applications may fill up the flash device at a faster rate than the rate at which those writes can be destaged to the SAN. This happens if SAN latency is high. If there is no space left on the flash device, the application cannot be allowed to write to the SAN because the SAN has stale data and overwriting stale data would cause data inconsistency and corruption. In this scenario, the application will stall until space can be made available on the flash device. A stalled application is clearly unacceptable; it would be preferable not to accelerate writes at all.

We introduce FVP, a write acceleration layer that uses host-side flash devices to accelerate VM writes in a clustered virtualized environment, while tolerating multiple cascading flash and host failures. FVP is a kernel module installed inside the ESX hypervisor [23] which intercepts VM I/Os and forwards them to the flash device. Since FVP sits inside the hypervisor, the I/O path from the guest OS to the flash device is short, resulting in good application performance. No change is required to the guest VM operating system.

FVP pools together the flash devices from all the hosts such that each host can access the data on another host's flash device. When a VM migrates to another host, its data on the previous host's flash device can be accessed by the new host eliminating inconsistency and data loss. VM mobility, DRS and HA are preserved.

VM writes are replicated to other hosts or peers. Therefore, in the event of a host or flash failure, other hosts co-ordinate with each other to destage their copy of VM writes to the SAN and restore VM consistency. By replicating VM writes, FVP can tolerate multiple host and flash failures. Recovery is distributed and happens without any central co-ordination. Our contributions are as follows:

1. FVP is a host-side write acceleration layer which supports transparent VM migration and seamlessly integrates into the virtualized environment.

2. FVP can handle multiple, cascading host and flash failures. To the best of our knowledge, ours is the only solution that provides fault tolerance. Recovery is distributed without requiring any coordination.

3. We introduce *Flow Control*, an I/O control mechanism designed to prevent write heavy applications from running out of flash space.

Though FVP has been implemented for ESX, the design principles are hypervisor independent and can be applied to any well-known hypervisor architecture, including Xen [24], Hyper-V [25], and KVM [26]. In the subsequent sections, we describe FVP in more detail.

2 Overview

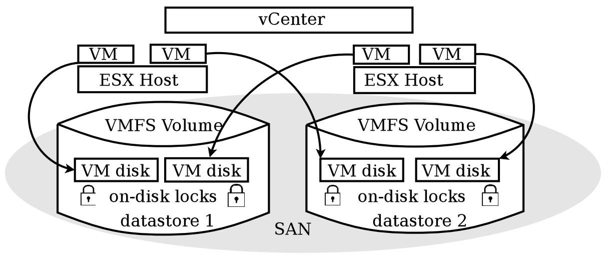


Figure 2: Virtualized Environment

Figure 2 depicts a typical virtualized environment with physical machines/hosts running the ESX hypervisor. A Storage Area Network (SAN) is used to consolidate storage and provide hosts shared access to it. A clustered file system, VMFS [27], provides multiple hosts simultaneous access to file system volumes or *datastores*. A datastore is a VMFS (block) based volume that houses VM virtual disks. A hypervisor cluster is a group of such hosts sharing one or more datastores via SAN. All the components and operations in the cluster are managed by a central entity, the *vCenter*.

A VM can only migrate to a host that has access to its datastore. Though a VM's files are accessible to every host in the cluster, VMFS moderates this access to one host alone. This is done using on-disk locks co-located on the same datastore as the VM's virtual disk. Only one host can acquire the VM's on-disk lock and this is the host that can write to the datastore on behalf of that VM. When DRS live migrates a VM, from one host to another, VMFS releases the on-disk lock for that VM. The new host now acquires the lock and runs the VM.

A FVP cluster is a hypervisor cluster in which every host runs FVP. VMs, thus, migrate only among FVP hosts. Henceforth, the term 'host' assumes that it is a FVP host, and 'cluster' assumes that all hosts in the cluster are FVP hosts.

3 VM Acceleration Policies

Each VM on an FVP host is configured with an *acceleration policy*. The acceleration policy dictates how VM writes and/or reads should be cached. Data center administrators can configure VMs with an appropriate write policy depending on VM workload.

Write Through (*wt*) policy: In *wt* [12, 13, 28] mode, FVP intercepts VM writes and issues them simultaneously to SAN and to flash. The write is acknowledged to the VM after it has been acknowledged by both the flash and the SAN. Typically, the longest time to acknowledgment is from the SAN, and the VM sees SAN latencies for writes.

Write-Back (no peers) (*wb*): For a VM in *wb*, FVP intercepts VM writes and issues them to the host's local flash alone. The write is acknowledged to the VM after flash completion. As writes accumulate on the flash device, they are batched and destaged/issued in batches to the SAN. Destaging is completely transparent to the VM. The VM, thus, sees flash latencies instead of SAN latencies for writes.

Write-Back with Peering (*wbp*): For a VM in *wbp*, FVP synchronously duplicates every write and sends it to another host, or *peer*, in the cluster while simultaneously writing it to local flash. When the peer writes VM data to its flash, an acknowledgment is sent back to the primary host housing the VM. FVP on the primary host then acknowledges the write to the VM. The peer holds on to the write until the primary has destaged it. Typically, data transmission over the network to the peer takes the longest. VM writes experience network *and* peer flash latencies.

FVP selects peer hosts based on rules/policies setup by administrators.
Datacenter administrators can setup fault domains that group together hosts based on the datacenter topology such as, hosts on a rack belong to one fault domain *etc.* Further, FVP also allows administrators to choose that hosts be configured with local peers within their fault domain and/or remote peers in other fault domains.

Un-cached: For a VM that is un-cached, FVP does not cache its data. Neither reads nor writes are accelerated.

For *wt*, *wb*, and *wbp* VMs reads are first issued to flash. In case of a cache miss, the read is issued to SAN and the data written to flash. Evictions, irrespective of caching policy are done on a LRU basis.

4 Seamless VM Migration

To balance resource utilization in datacenters, VMs are live migrated, or re-distributed across hosts [20]. An eligible host is one that has access to the VM's datastore. After migration, VMs continue to have access to their data because datacenters consolidate storage arrays under a shared SAN. However, host side flash is local to the host. If a VM having dirty data on its host's flash is migrated to another host, it loses access to that data. FVP solves this problem by enabling the new host to access the previous host's flash.

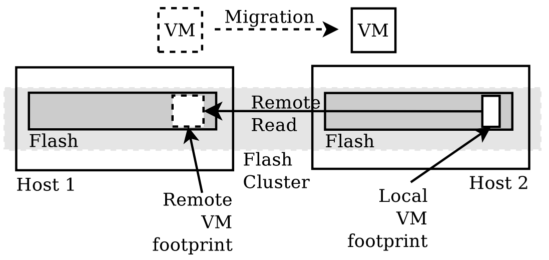


Figure 3: Virtual Machine Migration

Figure 3 illustrates how FVP orchestrates VM migration between two hosts. When a VM migrates from Host 1 to Host 2, its cached data on Host 1's flash, *i. e.*, its footprint, remains on Host 1. Host 1 is free to evict the VM's cache if it so requires.

FVP maintains state about the VM's previous host. This obviates the need for Host 2 to poll every other host to locate that VM's footprint. So, when the VM issues a read that causes a cache miss on Host 2's flash, Host 2 re-transmits the read to Host 1. Such a read is called a *remote read*. If the read request causes a cache miss on Host 1's flash (possibly because Host 1 has already evicted that data to reclaim flash space) Host 2 re-issues the read to the SAN.

In case of a cache hit on Host 1's flash, Host 1 transmits that data to Host 2. When Host 2 receives the data, it copies the data onto its own local flash. Thus, Host 2 begins building the VM's footprint locally. Owing to remote reads, VM reads that are a cache hit are faster, seeing only network and flash latency than if they were issued to SAN. Remote reads also alleviate traffic to the storage array while VM footprint is rebuilt on Host 2. By issuing remote reads for only what the VM requires, instead of eagerly copying the entire VM footprint, Host 2 conserves network bandwidth and flash wear.

The VM's footprint eventually rebuilds on Host 2's flash. When the number of cache misses for remote reads hits a pre-defined value, Host 2 stops issuing remote reads to Host 1. VM mobility is thus transparently preserved.

It would be ideal if a *wbp* VM were to migrate to one of its peers. The VM would then run off of its own footprint on the peer obviating the need to issue remote reads. However, FVP has no say over which host is chosen as the destination host for a VM. This is a decision made by the DRS process into which FVP has no visibility.

5 The Destager

The destager is a process that collects dirty VM writes cached on the flash device and issues them to the SAN. The writes are batched and all writes in a batch are issued concurrently. Batching is used to improve performance and ensure correctness as is described in this section. When the SAN acknowledges the writes, FVP marks those writes as destaged. Those writes can now be evicted from the cache. We discuss how the underlying storage fabric drives the design and behavior of the destager.

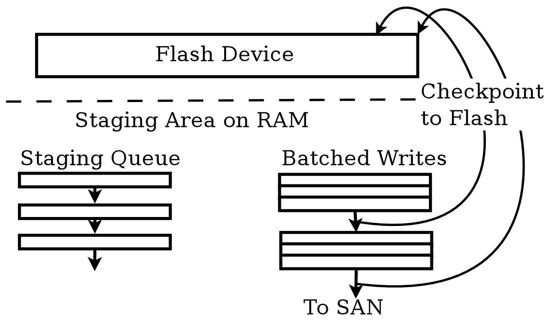


Figure 4: Workings of the destager

The sequence in which a storage controller may order concurrent writes is opaque to FVP, and indeed, to the application issuing such writes. This is not an issue for concurrent *non-overlapping* writes. However, concurrent overlapping writes, if not handled correctly, can cause application inconsistency and data corruption. Consider a write X at offset Y, denoted as X(Y). Consider two concurrent overlapping writes A(O) and B(O). FVP construes an ordering of A(O) followed by B(O) and issues them to flash. FVP only indexes the last write and therefore records B for offset O. So, when the VM issues a read for offset O it receives B. Meanwhile, the destager begins destaging the writes and both (A, O) and (B, O) being concurrent writes, issues them concurrently to the VM's datastore. The storage controller orders them (B, O) followed by (A, O), overwriting B with A. The datastore is now inconsistent with the VM.

To avoid such inconsistencies, a *gatekeeper*, assigns a monotonically increasing serial number (`ser#`) to *every* write. Non-overlapping writes tagged with a `ser#` are issued simultaneously to the flash. However, concurrent overlapping writes are serialized, *i. e.*, the gatekeeper waits for the flash device to acknowledge the write before issuing the next overlapping write. The writes are then enqueued in a staging queue in the *staging area*.

Figure 4 depicts the workings of the destager. The staging area is kept in RAM to save flash space and wear. It consists of a *staging* queue where writes are enqueued in order of their `ser#`. Note that the queue contains metadata only. The data resides on the flash device. The destager scans these writes, batches them up such that no two writes in a batch overlap. All writes in a batch are issued concurrently to the SAN. After the SAN acknowledges *all* those writes, the destager issues a `checkPoint` to the flash. A `checkPoint` is a special record consisting of the VM UUID, and `ser#` of the VM's last write record acknowledged by the SAN. FVP uses `checkPoint` records during failure recovery described in Section 7.

After the destager receives acknowledgment from the flash that it has completed writing the `checkPoint`, the destager proceeds to scan the staging queue again. If the VM was configured with peers, the `checkPoint` record is also transmitted to the peer/s. Each peer commits the `checkPoint` record to its flash. FVP on the primary and peer hosts is now free to evict from its flash device all records for that VM whose `ser#` $\leq$ `checkPoint` (`ser#`).

The storage controller may or may not order the writes in `ser#` order (indeed, it is unaware of any such metadata), but, since these writes are non-overlapping, writing them in any order it wishes does not lead to data corruption. The justification for such behavior is that even in the absence of FVP, for concurrent writes the storage controller gives the same consistency guarantees and therefore after the entire batch of writes has been committed to the SAN, the SAN is consistent with VM, though lagging by some time.

The staging queue is maintained on a per VM basis. The destager cycles through staging queues in a round robin fashion destaging one batch per queue before proceeding to the next VM's queue. To allow each VM a fair share of the SAN's bandwidth for servicing writes and reads, the size of each batch is capped to a configurable value. The maximum batch size, defined in the number of writes, is configured to match the queue depth of the storage's host bus adapter.

As a rule of thumb, issuing fewer but larger sets of concurrent writes to storage yields better throughput than issuing smaller and frequent concurrent writes. Therefore, deferring and consolidating writes while destaging yields better flash acceleration. A VM that issues frequent overlapping writes does not see the level of flash acceleration that another VM issuing fewer overlapping writes would see. If such writes are bursty, the flash is able to absorb the burst, simultaneously destage writes and catch up with the VM a few moments after the burst of writes has stabilized without degrading application throughput.

6 Flow Control

The flash space on an ESX host could potentially be shared by hundreds of VMs. Though every VM may have different space requirements which change over time [29], FVP implements a fair share policy when carving out flash space for individual VMs. Fair share, though an early implementation, has advantages such that it isolates VMs from any noisy neighbors. Noisy neighbors are VMs that claim a high proportion of flash space due to their large working sets. Consequentially, other VMs experience degraded performance. FVP implements fair share for its simplicity and to insulate VMs.

For sustained write bursts, a VM's writes accumulate on flash at a high rate filling up its quota of flash space. The destager works in tandem with the VM to clear up that flash space to accommodate new writes. However, the destager's throughput is predicated on the latency of the SAN. If SAN latency is high, during a sustained write burst, a VM's flash space fills up before the destager has a chance to catch up. The VM, then, would have to be stalled, *i. e.*, it cannot be allowed to issue I/Os until (a) the destager is able to reclaim space by flushing out accumulated writes and/or (b) FVP is able to evict cached *reads* from the flash device. When space is reclaimed, the VM can continue to issue I/Os, but if the write burst continues, the VM would have to be stalled once again to allow the destager to make more space and so on. At such times, the VM would experience degraded performance or SAN latencies.

The VM cannot be allowed to write to SAN because SAN has stale data. If the VM were allowed to write, and those writes overlapped with writes that are yet to be destaged, data corruption would occur.

To prevent VM performance degradation, FVP triggers a process called *flow control*. Flow control throttles the VM by introducing an artificial delay before a write is acknowledged back to the VM. Though this slows down the VM, it gives the destager some extra time to make space for new writes. The delay is calculated as a moving average over SAN latencies observed in the near past. FVP uses three heuristics to trigger flow control per VM:

1. The number of dirty VM writes.

2. The cumulative size of the dirty writes.

3. The expected time to destage those writes.

These were chosen because they are indicative of how fast the destager would be able make progress given its pending workload. A high number of writes in proportion to their cumulative size indicates that the VM is primarily issuing small writes. The destager's batch size being fixed, this means the destager has to cycle through a larger number of batches, and consequentially, engage in those many conversations with the SAN. A higher cumulative size of dirty writes in proportion to the number of dirty writes indicates that the VM is issuing large writes which typically incur higher SAN latencies. The expected time to destage all the writes is calculated using a moving average of SAN latencies seen in the past. Together, these three heuristics indicate the probability of whether the destager would be able to clear up flash space in time to accommodate VM writes. If any of the above heuristic counters cross pre-defined triggering values, flow control kicks in, progressively injecting increasing amounts of delay when acknowledging VM writes. Flow control slows down the incoming writes by introducing delays of the order of $1\times$ to up to $4\times$ the SAN latency to reduce the pressure on the staging area. However, if VM writes fill up its flash space because the destager is not able to make progress on account of a very slow SAN, the VM performance degrades to match that of the SAN.

Stalling is extremely rare. Only in the case of workloads that are write intensive over a prolonged period of time coupled with a very slow SAN would the VM be stalled.

Most VMs, however, are not continuously write intensive, but rather bursty [30]. Short write bursts are absorbed by the flash device and the VM throughput remains unaltered during the bursty period. Sustained write bursts may need to be throttled for the latter part of the bursty period, and in majority of the cases, continue to run while experiencing SAN latencies instead of flash latencies. Also, once the sustained burst of writes has lapsed the destager once again regains ground and the VM is freed from flow control.

Besides undesirable degraded VM performance, holding on to a large size of dirty VM data on the flash device increases the impact of data loss to a *wb* VM in case of a flash failure and the window of vulnerability in the case of host failure. Flow control, thus, prevents VM performance degradation, mitigates the consequences of failures, and speeds up VM migration.

7 Distributed Fault Tolerance

For *wb* VMs, in the event of a failure the SAN is left in an inconsistent state with respect to the VM in that some of the VM's writes have not yet been persisted to the SAN but have been acknowledged to the VM right after they were persisted to the host-side flash. The key challenge towards achieving crash consistent fault tolerance when using host-side flash devices for write back acceleration is ensuring that at the end of failure recovery cached VM writes are destaged correctly and completely to the SAN.

In the case of a host failure, for *wb* VMs, after the host recovers the destager flushes VM writes from its flash to the SAN from the last `checkPoint` onwards in order of their `ser#`. The `ser#` is persisted to flash as part of other metadata for every write. This ensures correctness, *viz.*, the writes are replayed in the order in which they were received by FVP.

However, HA may migrate affected *wb* VMs to another host while the failed host is recovering. If those VMs issue I/Os that overlap with previous I/Os that have not yet made it to the SAN, data corruption will occur.

To prevent data corruption, and co-ordinate the recovery process between two hosts as a *wb* VM migrates between them, FVP uses an on-disk lock file called `vault.lock`. For every VM, FVP persists its current acceleration policy and `checkPoint` record in the `vault.lock` file. The `vault.lock` file is located on the VM's datastore. Through atomic `vault.lock` access, VMFS arbitrates ownership of a VM's datastore between hosts such that the `vault.lock` file is locked and kept open by one host only. Only this host is eligible to execute I/Os on behalf of the VM to the SAN. The lock is held by the recovering host until recovery is complete.

In the case of a flash failure the cached writes are lost. FVP solves this problem by replicating VM writes onto peer hosts' flash devices. The peers can now flush those replicated writes to the SAN to complete recovery.

When a host loses connectivity to the storage fabric, the cached writes cannot be flushed to the SAN resulting in data loss. If any of the peers has access to the SAN, it can flush the replicated writes from its flash.

In addition to on-disk locks, hosts also monitor network, storage, and, peer health via regular heart beats. Thus, atomic access to the on-disk `vault.lock` file, read-only access to the `vault` file contents and heart beats together form the basis of FVP's failure recovery mechanism. FVP is designed to tolerate multiple flash, host, and network failures. Recovery is distributed; there is no master-slave protocol between hosts. We now discuss how this distributed failure recovery is instrumented for each of the failure scenarios.

7.1 Flash Failure

For a *wt* VM, in the event of a flash failure there is no data loss.

For a *wb* VM, in the event of a flash failure, if there was dirty data on the flash device, FVP stalls the VM. To

Failure	*wt*	*wb*	*wbp*
Flash	*wt* →un-cached No data loss	Data loss	Peer online replay *wb* →*wt* →un-cached
Host	No recovery required	Offline replay *wb* →*wt* →*wb*	Peer online replay *wbp* →*wt* →*wbp*
Network	NA	NA	Primary online replay *wbp* →*wt* →*wbp*
Multiple Failures	NA	For host failures, offline replay, or surrogate offline replay	Online/offline replay by primary, peer, or surrogate hosts

Table 1: Recovery from flash, host, network and multiple failures

avoid data corruption the VM cannot be allowed to write to the SAN. A flash failure for a *wb* VM results in data loss.

For a VM in *wbp*, in the event of a flash failure, to avoid data corruption FVP stalls the VM, and relinquishes lock on `vault.lock`. The peers, who periodically poll `vault.lock`, attempt to acquire the lock, contending against each other, where contention is resolved by VMFS. The peer that acquires the lock begins destaging VM writes starting from the last `checkPoint` that it had received from the primary host, regularly updating the `checkPoint` in `vault.lock`. After destaging is complete the peer updates `vault.lock` with the latest `checkPoint`, a cache policy of *wt*, and relinquishes lock on `vault.lock`. The process of destaging writes by a *live* host in the event of a failure is called *Online Replay*.

The VM policy of *wt* indicates to the peers that the VMs dirty data has been flushed. They drop all of the VM's writes from their flash and stop polling `vault`. The host that destages VM writes updates `vault.lock` `checkPoint` regularly so as to communicate to the remaining peers, via `vault`, that they can clear up writes that have already been destaged. Also, in case of peer failure or peer flash failure, the remaining peers can pick up where the last peer left off by starting with the `checkPoint` in `vault.lock` minimizing recovery time.

When the primary host detects from `vault` that the VM has transitioned to *wt*, it reacquires `vault.lock`, updates `vault.lock` with an acceleration policy of 'un-cached', and, un-stalls the VM.

7.2 Host Failure

There is no data loss for a *wt* VM in the case of a host failure.

In the event of a host failure all the `vault.lock` files for the affected VMs are released as part of the failure detection process. After the host recovers, the host reacquires those locks. For a *wb* VM, after the failed host recovers, FVP scans all the cached writes on the flash, building an inventory of resident *wb* VMs and their dirty data. This scan is necessary because as a consequence of host failure, the 'in RAM' staging data structures used during normal destaging and online replay operations are no longer available.

As soon as the scan is complete, the host and FVP come back online, and the VMs are powered on. VMs having dirty data on flash are stalled until their data is destaged, but FVP is ready to service VMs not affected by the host failure. For VMs with dirty data, FVP tries to lock `vault.lock`. For those VMs whose `vault.lock` was acquired, their dirty writes are destaged in order of `ser#` while regularly updating `vault.lock` `checkPoint`. At the end of replay, each `vault.lock` is updated with a cache policy of *wt*. This process of destaging records by a host recovering from a failure is called *Offline Replay*.

If HA has migrated any *wb* VMs to another host before the previous host has recovered, the new host is now able to acquire a lock on their `vault.lock` file and is, therefore, now eligible to issue I/Os on behalf of the migrated VM. This could cause data corruption. To prevent this from happening, FVP persists VM acceleration policy in `vault.lock`. When the new host acquires `vault.lock` for a migrated VM, it gleans that the VM was in *wb* on the previous host. This indicates to the new host that the VM has pending writes on the previous host's flash which have not yet been flushed to the SAN. It releases `vault.lock` and stalls the VM.

A read only copy of `vault.lock` is kept in another file called `vault`. The new host polls `vault` periodically so it can re-acquire `vault.lock` when the previous host is done destaging.

For *wbp* VMs, while the primary host is down, the peer/s detect host failure due to missing heart beats. Either that, or the peer/s detect host failure because one of them is able to acquire `vault.lock` for affected VMs. The peer that succeeds, executes an online replay on behalf of those VMs regularly updating the `checkPoint` in `vault.lock`. At the end of online replay `vault.lock` is updated with the last VM `checkPoint`, and *wt* cache policy. The other peers, on gleaning transition of the VM to *wt*, drop all data belonging to that VM from their flash.

The peers no longer participate in providing fault tolerance for the VM.

Meanwhile, FVP on the recovering primary host tries to acquire `vault.lock`, and fails. When it eventually acquires `vault.lock` it detects that the VM has transitioned to *wt*. No offline replay is required for this VM. The VM is un-stalled, new peers are configured and it transitions back to running in *wbp*.

7.3 Network Failure

A network failure is one when peers lose connectivity with each other. This only affects VMs in *wbp*. In case of a network failure, writes can no longer be replicated to peers. When FVP on the primary host detects a network failure, it initiates a $wb \rightarrow wt$ transition of the affected VMs. The destager flushes all the dirty data for those VMs. Flow control is invoked to allow the destager to make quick progress. During this stage, the affected VMs continue running in *wb*, but without the desired level of write redundancy. When a very small number of writes remain to be destaged, the VM is stalled, the remaining writes are destaged, the VM is un-stalled, released from flow control, and the transition to *wt* is complete.

The peers glean the transition to *wt* from `vault` and drop all cached data for the concerned VMs. For every affected VM, FVP chooses new peers from a list of other candidate hosts. Once the desired number of peers are re-established the VM is transitioned back to *wbp*.

It is possible that after the failed peer comes back up, the VM has already transitioned to *wb*/*wbp*. The peer may then incorrectly deduce that it should continue holding on to the VMs dirty data on its flash. To prevent this, FVP maintains a generation number (`gen#`) for every VM which is also persisted in `vault.lock`. The `gen#` is a monotonically increasing number that keeps track of a VM's transitions through cache policies. It is incremented every time a VM transitions from $wb \rightarrow wt$ or $wbp \rightarrow wt$. The incremented `gen#` indicates to the freshly recovered peer that the VM's data it contains on its flash has already been replayed. It is now free to evict that data.

Lastly, a network failure, could be misconstrued by peers, as a primary host failure. They try to acquire `vault.lock` while monitoring `vault` and take over replay if the primary host fails.

7.4 SAN Failure

A SAN failure is when hosts are unable to connect to the storage array. It is possible that such disruption affects only partial hosts. If FVP on the primary host has lost connectivity to the storage array, it stalls the VMs. For VMs in *wbp*, the primary host depends on the peers to replay dirty data. If only the peers have lost connectivity to storage, they begin to fail remote writes sent for replication by the primary host. This indicates to the primary host that the peers are no longer able to keep replicas. The VMs are flow controlled, their data is destaged and then transitioned to *wb*. To summarize, any host that has access to the storage array can acquire `vault.lock` and replay dirty data. Note that even if all hosts have experienced SAN failure, data on the flash device is still intact.

7.5 Multiple Cascading Failures

If during an online or offline replay, the concerned host fails, the remaining replay can be completed either by its peers or itself on recovery. If the primary host *and* peers fail and are unable to recover, the flash device can be re-installed on another host that has access to the VM's datastore. This is possible because all necessary metadata required to conduct replay is persisted on the flash device itself. The surrogate host would scan the flash device, acquire the necessary `vault.lock` from the VM's datastore and then complete replay for affected VMs. To minimize recovery time in the case of cascading failures, `vault.lock` (and consequently, `vault`) are regularly updated with the last `checkPoint`.

7.6 Distributed Recovery

The key to distributed recovery is access to shared storage and exclusive ownership of `vault.lock` among participating hosts. Shared storage access enables primary and peer hosts to monitor VM acceleration policy transitions and destaging progress. VMFS ensures access to the on-disk lock is atomic preventing any split-brain scenarios allowing hosts to co-ordinate failure recovery one at a time via online/offline replay picking up from where the last host left off. For instance, in the case of primary host failure, one of the peers takes over online replay. If this peer fails, the next peer may take over. If both peers fail and the primary host comes back online, the remaining data, is destaged by the primary host via offline replay.

Table 1 summarizes how FVP recovers from various failure scenarios. FVP recovery is crash consistent. Data corruption is prevented by use of `checkPoint`, `gen#` and `ser#`. Recovery is complete when every write acknowledged to the VM before failure/crash is committed to the SAN. Hence, after recovery, affected VMs return to a crash consistent state. VMs in *wt* are always crash consistent. For VM in *wb*, in case of a flash failure, there is data loss. VMs in *wbp* are protected against p flash, and $p+1$ host failures, where p is the number of peers.

8 Evaluation

Our setup consists of two hosts, each a HP Proliant DL 380 G6, 8 cores, ®Intel Xeon CPU E5540 at 2.533 GHz, 55 GB RAM, and 120 GB flash drive. Each host runs VMware vSphere 5.5 Enterprise Plus. The shared storage is a storage appliance with disk spindles and a flash cache. We will show that even when used with this faster than average SAN, FVP provides significant speed and latency improvement. With a slower HDD-based SAN, latency improvements will be more significant.

8.1 Short Write Bursts

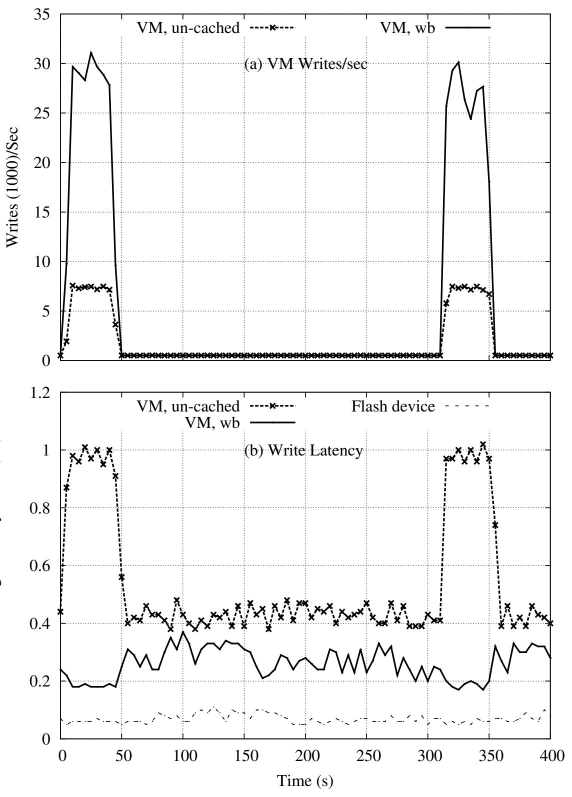


Figure 5: Short Write Bursts: (a) VM Writes/sec, (b) Write Latency

The objective of the first experiment is to demonstrate how FVP absorbs *short* write bursts thereby masking the VM from latency spikes of the SAN. To do this, we generated a workload resembling writes issued by a database servicing an OLTP application [31]. The workload, generated using Iometer [32], issued a burst of sequential writes for a short duration (40 seconds) followed by a slow and steady pace of random writes for 280 seconds. The short bursts of sequential writes simulated writes generated by a database as a result of multiple transaction commits, and log flushing. The steady stream of random writes simulated inserts/updates to support short database transactions. All writes were 8 KB in size, and 8 KB aligned.

To illustrate the benefits of using FVP to accelerate such workloads, we compared VM performance in *wb* and in an un-cached mode. The workload was run twice, once with the VM configured in *wb* and next with the VM in un-cached mode. In the un-cached mode, writes were not cached on the flash device but were issued directly to the SAN.

Figure 5 (a) compares the rate at which writes were acknowledged to the VM when running in *wb* with that when the VM ran in un-cached mode. In un-cached mode, writes were acknowledged to the VM after they were written to the SAN, at about 7,500 writes/second. In *wb*, during the first write burst (between 0 and 50 seconds) FVP acknowledged the writes to the VM as soon as the writes issued to the flash device were completed, at the rate of about 30,000 writes/second. Even though the SAN was slower to acknowledge writes during the bursty period, the VM did not see a degradation in performance when running in *wb*. This is because the write burst was absorbed by the flash device and the VM writes were acknowledged at flash speed. In the background, the destager issued those accumulated writes to the SAN.

Next, Iometer issued a steady stream of random writes at a slower rate for 280 seconds. During this period, as seen in Figure 5 (a), VM writes were acknowledged at the same rate in *wb* and in the un-cached mode. This is because, the incoming rate of writes was slow enough so that FVP and the SAN were able to service all the writes in a batch before the next batch of writes was issued.

This cycle of short bursts followed by steady writes was repeated once more.

Figure 5 (b) plots the average write latency, *i. e.*, the time from write issue to write completion, as observed by the VM. The figure also plots the flash write latency when VM writes were cached (*wb*). Together, the Figures 5 (a) and (b) demonstrate two key strengths of FVP: the first being that during the bursty period, the VM latency in *wb* tracked flash latency, *not* SAN latency. This allowed the VM to issue 4× the number of writes during the bursty period in *wb* than when un-cached. The second, is that the write latencies in *wb* were steady *and* low. In contrast, write latencies observed by the un-cached VM varied from 0.4 ms at best, to 1 ms during the bursty period.

8.2 Sustained Write Bursts

Figures 6 (a) and (b), demonstrate how FVP handles *sustained* write bursts. Figure 6 (a) depicts the rate at which writes were acknowledged to the VM by FVP and to the destager by the SAN, while Figure 6 (b) depicts VM, and

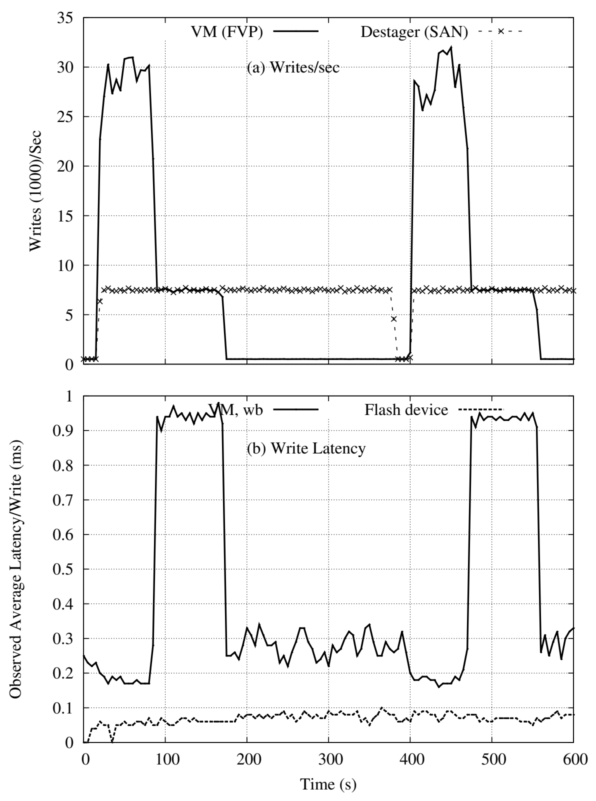


Figure 6: Sustained Write Bursts: (a) Writes/sec, (b) Write Latency

flash write latencies for those corresponding times.

To demonstrate how flow control manifests, we generated a workload similar to the OLTP workload, but quadrupled the write burst period (160 seconds). For the first 80 seconds, writes were acknowledged to the VM at flash speed. After 80 seconds, due to the volume and rate at which writes accumulated on flash, FVP triggered flow control. Flow control introduced delays in the write completion path before the writes were acknowledged to the VM. The induced delays were equal to the SAN's latency. Hence, VM writes experienced SAN latencies. These additional delays are shown in the Figure 6 (b) where VM latencies increased over and above flash latencies during the flow control period. Flash latencies were always steady at about 70 μseconds.

Once the sustained burst was over, the rate of incoming writes dropped. This allowed the destager to catch up and destage the pending records. The VM was then released from flow control. We know it was released, because at the beginning of the next burst, the VM acknowledged writes at flash speed again.

SAN latencies may increase if/when the SAN is overloaded because of the cumulative I/Os issued by several VMs or due to SAN administrative tasks. This too can trigger flow control. By using flow control, FVP avoids stalling applications/VMs which would unacceptably interrupt business processes. Instead VMs continue to run gracefully at SAN speed.

8.3 Read Latencies during VM migration

The objective of the next experiment is to demonstrate how FVP preserves VM mobility with minimal impact to VM performance. Figure 7 shows latencies observed by a VM running Iometer while issuing random 4K reads. For the first 700 seconds, the VM experiences millisecond latencies as the data is fetched from SAN. This was because, none of the data was cached before the workload was started. As the cache was populated and hits increased, read latencies gradually reduced. Once the working set is cached, the VM experiences flash latencies of the order of 450 μseconds. Then the VM is migrated to another host. In response to cache misses the new host begins to issue reads to the previous host and gradually builds up the VM's footprint on its local flash. The increased read latencies in the graph after VM migration when the new host issues remote reads are due to the additional latency incurred in transmitting the read data over the network from the previous host to the new one. As the new host gradually builds up the VM's footprint, fewer remote reads are issued. This can be seen from the graph where read latencies gradually reduce after migration. Once the VM footprint on the new host's flash is complete, remote reads are no longer issued and the VM experiences flash latencies once again.

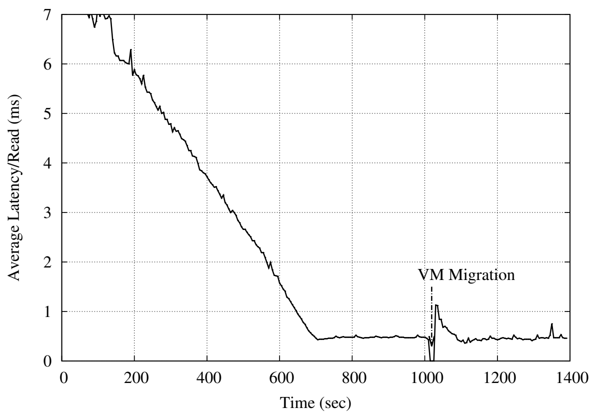


Figure 7: VM observed read latencies after migration

8.4 Fault Tolerance Cost vs Benefit

The objective of the final experiment is to analyze the trade-off between fault tolerance and performance. To do this, we compare VM throughput in *wt* with that in

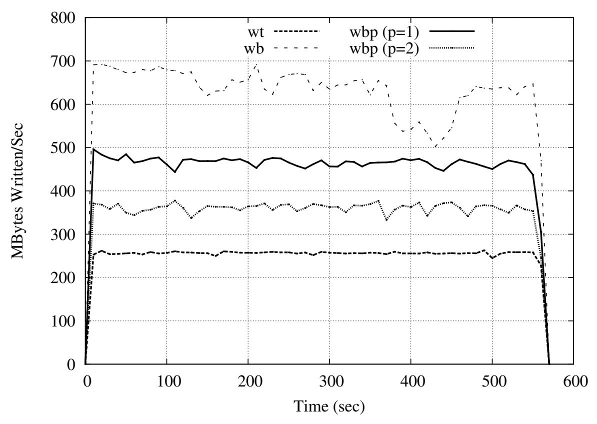


Figure 8: Combined throughput of two VMs in *wt*, *wb*, *wbp* (p=1), *wbp* (p=2). Workloads: Microsoft Exchange Server Jetstress and fio

	fio Throughput	JetStress Transactional IOPs
wb	468.81 MB/s	9325
wbp (p=1)	322.48 MB/s	7677
wbp (p=2)	235.32 MB/s	6605
wt	149.44 MB/s	5514

Table 2: Application performance in *wt*, *wb*, *wbp* (p=1), *wbp* (p=2)

wb, *wbp* ($p=1$), and *wbp* ($p=2$). Figure 8 depicts the combined throughput of two VMs. The guest OS of one VM was Windows 2008, running Microsoft Exchange Server Jetstress [17]. Jetstress simulates the workload of an Exchange database consisting of database transactions (reads and writes), log writes, and maintenance tasks such as database compaction, defragmentation and checksums. Jetstress was configured with 150 mailboxes allocated over 40 GB. The guest OS of the second VM was Ubuntu running fio [18], an IO workload generator. fio was configured to simulate a throughput intensive workload with two threads, each thread issuing 64 KB sequential writes with 8 writes in flight.

The first observation from Figure 8 is that write caching (*wb*) has clearly accelerated VM throughput, a definite gain over *wt*. The second observation is that when writes were replicated to the peer (*wbp*), the VM throughput slowed down compared to that in *wb*. In *wb*, the average throughput of the VMs was around 600 MB/s, while in *wbp* ($p=1$) it was 475 MB/s. This difference was because in *wbp*, every write incurred an additional latency when it was replicated across the network onto the peer's flash.

DRS and HA use the network interconnect between hosts to migrate VMs between them. FVP uses the same network to transmit writes between a host and its peers for fault tolerance. The cumulative throughput for *wbp* ($p=2$) was lesser than that with *wbp* ($p=1$) because currently the FVP network stack instrumentation does not exploit multiple NICs. This work is, however, planned for the coming future. With multiple NIC support, VM throughput would track that of the slowest peer network.

Table 2 lists the performance of the individual applications in *wt*, *wb*, and *wbp* ($p=1, p=2$). For fio, the table lists the average throughput whereas for JetStress, the table lists the transaction rate (IOPs). Figure 8 and Table 2, together illustrate that the cost of replicating writes is reduced VM throughput. However, the throughput of a *wbp* VM is still better than a *wt* VM with the added advantage of fault tolerance. To summarize, FVP solves fault tolerance by replicating writes, *and* achieves write acceleration by using host-side flash.

9 Related Work

Holland *et al.* [13] and Byan *et al.* [12] prescribe using host side flash for *wt* only because *wb* causes consistency issues with VM migration. Byan *et al.* explore various options for deploying host-side flash: integrated with the storage controller, the network, the hypervisor *etc.*, and choose to deploy their solution within the hypervisor.

Koller *et al.* [11] discuss the trade-offs of using host-side flash with respect to data consistency, staleness and performance for *wt* and *wb*. They propose ordered and journaled destaging. Ordered destaging evicts writes in the order in which they were issued, like FVP. In addition, Koller *et al.* parallelize evictions for unrelated writes. Journaled destaging coalesces writes to absorb write bursts. Using application specified hints Journaled destaging provides application level consistency. FVP also offers a best-effort application level consistent destager, but for the sake of brevity, and to focus on our key contributions, we have not elaborated on it.

Qin *et al.* [14] use application specified write barriers to achieve application consistency when using host-side flash to accelerate writes. Application specified write barriers, or hints are not distinguishable to the FVP kernel module. This was a deliberate decision; one that allows FVP to seamlessly integrate into the virtualized environment. Further, in an enterprise environment third party softwares [33] are employed to perform backups. These software quiesce the guest OS to allow for application consistent snapshots. FVP seamlessly detects this activity and transitions those *wb* VMs to *wt* for the duration of the snapshot operation. The details of this mechanism have not been discussed in the paper since that is not the main focus. In addition FVP also provides datacenter administrators a switch to initiate a *wb* $\rightarrow$ *wt* transition on VMs for the duration of the backup/snapshot window.

Koller *et al.* also discuss the cost in terms of the potential data loss that might be incurred in the case of failure for applications when they use *wb* acceleration. With *wbp* in FVP VMs are protected against p flash failures and $p+1$ host failures to minimize the probability of a data loss. The cost, however, is increased recovery time for when the affected VMs are stalled until their *wb* data has been flushed to the storage.

How long a VM is kept stalled depends on several multi-dimensional factors. The most critical being the volume of dirty writes. The time required to destage those writes, and therefore, the period for which the VM is kept stalled is proportional to the size of staged data. The other factors being the SAN speed/performance, the saturation of the SCSI network, the load on the disk array, the number of VMs being hosted, the workload those VMs are generating *etc.* These factors, other than staged data size, change dynamically depending on the workload supported by the datacenter. If these were steady, then the time for a which a VM needs to be stalled is proportional to the amount of staged data that needs to be flushed. To mitigate the cost of recovery, FVP caps the size of dirty data that can accumulate for a VM. As discussed in 6, flow control is invoked to moderate VM write footprint. To tune this further, datacenter administrators can configure the staging size based on the RPO requirements for applications.

Also, in the case of FVP, recovery speeds up when peers are involved. When a host fails, the peers can finish destaging so that when the VMs are brought back up most or all of their data has been already flushed to SAN. Thus, peering reduces the cost of recovery for *wbp* VMs.

10 Future Work

We are working on building more intelligence into FVP; an adaptive resource manager that detects VM workload characteristics and priority, and in response, tunes VM flash space usage, acceleration policy, eviction policies, and destager behavior for that VM.

11 Conclusion

FVP brings seamless, fault tolerant write acceleration using host-side flash to HA and DRS enabled virtualized datacenters. Failure recovery is distributed and, with p peers, is $p+1$ host/flash failure tolerant. FVP absorbs short write bursts so VMs see flash latencies instead of degraded SAN latencies during the bursty period. This masks VMs from SAN latency spikes, improves VM performance predictability to help deliver SLA objectives allowing IT teams to accelerate write heavy applications, such as databases and Virtual Desktop Infrastructure [34]. For sustained write bursts FVP uses flow control; write intensive applications continue without stalling. The storage can now be provisioned linearly with new hosts, instead of being provisioned for bursty workloads. FVP helps increase VM density allowing existing hosts to support more VMs without having to provision additional storage. This increase in performance means happier end users and, consequently, fewer support calls relating to poor application performance. This allows IT organizations to consolidate more hosts and economize.

12 Acknowledgments

Heiko Köhler, Venkatesh Kothakota, and Kaustubh Patil helped implement crash consistent write back acceleration. Sameer Narkhede implemented flow control. Satyam Vaghani, Akhilesh Joshi, Eric Cohen, Alex Depoutovitch, Nakul Dhotre, and Bala Narasimhan provided feedback during the preparation of this paper. Govindarajan Soundararajan conducted experiments to evaluate the performance of remote reads after VM migration. We thank our anonymous reviewers and our shepherd, Ashvin Goel for valuable feedback.

References

[1] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir, "ELI: Bare-metal Performance for I/O Virtualization," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, (London, England, UK), pp. 411–422, ACM, 2012.

[2] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman, "The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM," *ACM SIGOPS Operating Systems Review*, vol. 43, pp. 92–105, Jan. 2010.

[3] J. Shafer, "I/O Virtualization Bottlenecks in Cloud Computing Today," in *Proceedings of the 2Nd Conference on I/O Virtualization*, WIOV'10, (Pittsburgh, PA, USA), pp. 5–5, USENIX Association, 2010.

[4] NetApp, Inc., "Flash Cache - Clear Flash Cache - Improve Storage Performance." {http:

//www.netapp.com/us/products/storage-systems/flash-cache/}.

[5] SanDisk Corporation, "Hybrid Flash Storage Appliance :: ioControl :: Fusion-io." {http://www.fusionio.com/products/iocontrol/}.

[6] EMC Corporation, "EMC VNXe3200 Hybrid Storage - EMC Store." https://store.emc.com/Product-Family/EMC-VNXe-Products/EMC-VNXe3200-Hybrid-Storage/p/VNE-VNXe3200-Hybrid-Storage.

[7] Nimble Storage, "Data Storage Solutions — Enterprise Information Management — Flash-Based Storage - Nimble Storage." http://www.nimblestorage.com/.

[8] Saxena, Mohit and Swift, Michael M. and Zhang, Yiying, "FlashTier: A Lightweight, Consistent and Durable Storage Cache," in *Proceedings of the 7th ACM European Conference on Computer Systems (Eurosys '12)*, (Bern, Switzerland), pp. 267–280, 2012.

[9] Pure Storage Inc., "Flash array storage - all-flash enterprise array." {http://www.purestorage.com/flash-array/}.

[10] EMC Corporation, "XtremIO — All-Flash Scale-Out Enterprise Storage Arrays." http://www.xtremio.com/.

[11] R. Koller, L. Marmol, R. Rangaswami, S. Sundararaman, N. Talagala, and M. Zhao, "Write Policies for Host-side Flash Caches," in *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, (San Jose, CA), pp. 45–58, USENIX, 2013.

[12] S. Byan, J. Lentini, A. Madan, and L. Pabon, "Mercury: Host-side flash caching for the data center," in *Proceedings of the IEEE 28th Symposium on Mass Storage Systems and Technologies*, (Pacific Grove, CA), pp. 1–12, 2012.

[13] D. A. Holland, E. Angelino, G. Wald, and M. I. Seltzer, "Flash caching on the storage client," in *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, (San Jose, CA), pp. 127–138, USENIX, 2013.

[14] D. Qin, A. D. Brown, and A. Goel, "Reliable write-back for client-side flash caches," in *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC 14)*, (Philadelphia, PA), pp. 451–462, USENIX Association, June 2014.

[15] SanDisk Corporation, "ioTurbine :: Fusion-io." {http://www.fusionio.com/products/ioturbine/}.

[16] EMC Corporation, "EMC XtremCache Server Flash Cache." {http://www.emc.com/storage/xtrem/xtremcache.htm}.

[17] Microsoft Corporation, "Microsoft Exchange Server Jetstress 2013 Tool." http://www.microsoft.com/en-us/download/details.aspx?id=36849.

[18] Jens Axboe, "fio - Flexible I/O Tester Synthetic Benchmark." http://freecode.com/projects/fio.

[19] M. Nelson, B.-H. Lim, and G. Hutchins, "Fast transparent migration for virtual machines," in *Proceedings of the 2005 USENIX Annual Conference (USENIX ATC 2005)*, (Anaheim, CA), pp. 391–394, USENIX, 2005.

[20] VMWare, Inc., "VMWare Distributed Resource Scheduler (DRS)." {http://www.vmware.com/files/pdf/VMware-Distributed-Resource-Scheduler-DRS-DS-EN.pdf}.

[21] VMWare, Inc., "VMWare Distributed Power Management." {http://www.vmware.com/files/pdf/Distributed-Power-Management-vSphere.pdf}.

[22] VMWare, Inc., "VMWare High Availability (HA)." {http://www.vmware.com/files/pdf/VMware-High-Availability-DS-EN.pdf}.

[23] VMWare, Inc., "VMWare ESX and VMWare ESXi." http://www.vmware.com/files/pdf/VMware-ESX-and-VMware-ESXi-DS-EN.pdf.

[24] The Linux Foundation, "The Xen Project, the powerful open source industry standard for virtualization." http://www.xenproject.org/.

[25] Microsoft Inc., "Virtualization for your modern datacenter and hybrid cloud." http://www.microsoft.com/en-us/server-cloud/solutions/virtualization.aspx.

[26] RedHat , "Kernel-based Virtual Machine." http://www.linux-kvm.org/page/Main_Page.

[27] S. B. Vaghani, "Virtual Machine File System," *ACM SIGOPS Operating Systems Review*, vol. 44, pp. 57–70, December 2010.

[28] M. Srinivasan, "Flashcache : A Write Back Block Cache for Linux." {https://github.com/facebook/flashcache/}.

[29] C. A. Waldspurger, “Memory Resource Management in VMware ESX Server,” *SIGOPS Operating System Review*, vol. 36, pp. 181–194, Dec. 2002.

[30] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller, “Measurement and Analysis of Large-scale Network File System Workloads,” in *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC’08, (Berkeley, CA, USA), pp. 213–226, USENIX Association, 2008.

[31] J. Gray, *Benchmark Handbook: For Database and Transaction Processing Systems*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992.

[32] “Iometer.” {`http://www.iometer.org/`}.

[33] Veeam, Inc., “Veeam: Availability for the Modern Data Center.” `http://www.veeam.com/`.

[34] Wikipedia, “Desktop Virtualization.” `http://en.wikipedia.org/wiki/Desktop_virtualization`.

B*e*trFS: A Right-Optimized Write-Optimized File System

William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet*, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh*, Michael Bender, Martin Farach-Colton†, Rob Johnson, Bradley C. Kuszmaul‡, and Donald E. Porter

*Stony Brook University, *Tokutek Inc., †Rutgers University, and ‡Massachusetts Institute of Technology*

Abstract

The ***B*$^\varepsilon$*-tree File System***, or B*e*trFS, (pronounced "better eff ess") is the first in-kernel file system to use a write-optimized index. Write optimized indexes (WOIs) are promising building blocks for storage systems because of their potential to implement both microwrites and large scans efficiently.

Previous work on WOI-based file systems has shown promise but has also been hampered by several open problems, which this paper addresses. For example, FUSE issues many queries into the file system, superimposing read-intensive workloads on top of write-intensive ones, thereby reducing the effectiveness of WOIs. Moving to an in-kernel implementation can address this problem by providing finer control of reads. This paper also contributes several implementation techniques to leverage kernel infrastructure without throttling write performance.

Our results show that B*e*trFS provides good performance for both arbitrary microdata operations, which include creating small files, updating metadata, and small writes into large or small files, and for large sequential I/O. On one microdata benchmark, B*e*trFS provides more than 4× the performance of `ext4` or `XFS`. B*e*trFS is an ongoing prototype effort, and requires additional data-structure tuning to match current general-purpose file systems on some operations such as deletes, directory renames, and large sequential writes. Nonetheless, many applications realize significant performance improvements. For instance, an in-place `rsync` of the Linux kernel source realizes roughly 1.6–22× speedup over other commodity file systems.

1 Introduction

Today's applications exhibit widely varying I/O patterns, making performance tuning of a general-purpose file system a frustrating balancing act. Some software, such as virus scans and backups, demand large, sequential scans of data. Other software requires many small writes (microwrites). Examples include email delivery, creating lock files for an editing application, making small updates to a large file, or updating a file's `atime`. The underlying problem is that many standard data structures in the file-system designer's toolbox optimize for one case at the expense of another.

Recent advances in write-optimized indexes (WOI) [4, 8–10, 23, 27, 28] are exciting because they have the potential to implement both efficient microwrites and large scans. The key strength of the best WOIs is that they can ingest data up to two orders of magnitude faster than B-trees while matching or improving on the B-tree's point- and range-query performance [4, 9].

WOIs have been successful in commercial key-value stores and databases [2, 3, 12, 17, 20, 33, 34], and previous research on WOIs in file systems has shown promise [15, 25, 31]. However, progress towards a production-quality write-optimized file system has been hampered by several open challenges, which we address in this paper:

- **Code complexity.** A production-quality WOI can easily be 50,000 lines of complex code, which is difficult to shoehorn into an OS kernel. Previous WOI file systems have been implemented in user space.
- **FUSE squanders microwrite performance.** FUSE [16] issues a query to the underlying file system before almost every update, superimposing search-intensive workloads on top of write-intensive workloads. Although WOIs are no worse for point queries than any other sensible data structure, writes are much faster than reads, and injecting needless point queries can nullify the advantages of write optimization.
- **Mapping file system abstractions onto a WOI.** We cannot realize the full potential performance benefits of write-optimization by simply dropping in a WOI as a replacement for a B-tree. The schema and use of kernel infrastructure must exploit the performance advantages of the new data structure.

This paper describes the ***B*$^\varepsilon$*-tree File System***, or B*e*trFS, the first in-kernel file system designed to take full advantage of write optimization. Specifically, B*e*trFS is built using the mature, well-engineered B^ε-tree implementation from Tokutek's Fractal Tree index, called TokuDB [33]. Our design is tailored to the performance characteristics of Fractal Tree indexes, but otherwise uses them as a black-box key-value store, so many of our design decisions may be applicable to other write-optimized file systems.

Experiments show that B*e*trFS can give up to an order-of-magnitude improvement to the performance of file creation, random writes to large files, recursive directory traversals (such as occur in `find`, recursive `grep`s, backups, and virus scans), and meta-data updates (such as updating file `atime` each time a file is read).

The contributions of this paper are:

- The design and implementation of an in-kernel, write-optimized file system.
- A schema, which ensures both locality and fast writes, for mapping VFS operations to a write-optimized index.
- A design that uses unmodified OS kernel infrastructure, designed for traditional file systems, yet minimizes the impact on write optimization. For instance, B*e*trFS uses the OS kernel cache to accelerate reads without throttling writes smaller than a disk sector.
- A thorough evaluation of the performance of the B*e*trFS prototype. For instance, our results show almost two orders of magnitude improvement on a small-write microbenchmark and a $1.5\times$ speedup on applications such as `rsync`

Our results suggest that a well-designed B^ε-tree-based file system can match or outperform traditional file systems on almost every operation, some by an order of magnitude. Comparisons with state-of-the-art file systems, such as `ext4`, `XFS`, `zfs`, and `btrfs` support this claim. We believe that the few slower operations are not fundamental to WOIs, but can be addressed with a combination of algorithmic advances and engineering effort in future work.

2 Motivation and Background

This section summarizes background on the increasing importance of microwrites, explains how WOIs work, and summarizes previous WOI-based file systems.

2.1 The microwrite problem

A microwrite is a write operation where the setup time (i.e. seek time on a conventional disk) exceeds the data-transfer time. Conventional file-system data structures force file-system designers to choose between optimizing for efficient microwrites and efficient scans.

Update-in-place file systems [11, 32] optimize for scans by keeping related items, such as entries in a directory or consecutive blocks in a file, near each other. However, since items are updated in place, update performance is often limited by the random-write latency of the underlying disk.

B-tree-based file systems store related items logically adjacent in the B-tree, but B-trees do not guarantee that logically-adjacent items will be physically adjacent. As a B-tree *ages*, leaves become scattered across the disk due to node splits from insertions and node merges from deletions. In an aged B-tree, there is little correlation between the logical and physical order of the leaves, and the cost of reading a new leaf involves both the data-transfer cost and the seek cost. If leaves are too small to amortize the seek costs, then range queries can be slow. The seek costs can be amortized by using larger leaves, but this further throttles update performance.

At the other extreme, logging file systems [5, 7, 26, 29, 30] optimize for writes. Logging ensures that files are created and updated rapidly, but the resulting data and metadata can be spread throughout the log, leading to poor performance when reading data or metadata from disk. These performance problems are particularly noticeable in large scans (recursive directory traversals and backups) that cannot be accelerated by caching.

The microwrite bottleneck creates problems for a range of applications. HPC checkpointing systems generate so many microwrites that a custom file system, PLFS, was designed to efficiently handle them [5] by exploiting the specifics of the checkpointing workload. Email servers often struggle to manage large sets of small messages and metadata about those messages, such as the `read` flag. Desktop environments store preferences and active state in a key-value store (i.e., a registry) so that accessing and updating keys will not require file-system-level microdata operations. Unix and Linux system administrators commonly report 10–20% performance improvements by disabling the `atime` option [14]; maintaining the correct `atime` behavior induces a heavy microwrite load, but some applications require accurate `atime` values.

Microwrites cause performance problems even when the storage system uses SSDs. In a B-tree-based file system, small writes trigger larger writes of entire B-tree nodes, which can further be amplified to an entire erase block on the SSD. In a log-structured file system, microwrites can induce heavy cleaning activity, especially when the disk is nearly full. In either case, the extra write activity reduces the lifespan of SSDs and can limit performance by wasting bandwidth.

2.2 Write-Optimized Indexes

In this subsection, we review write-optimized indexes and their impact on performance. Specifically, we describe the B$^\varepsilon$-tree [9] and why we have selected this data structure for B*ε*trFS. The best WOI can dominate B-trees in performance rather than representing a different trade-off choice between reads and writes.

B$^\varepsilon$-trees. A B$^\varepsilon$-tree is a B-tree, augmented with per-node buffers. New items are inserted in the buffer of the root node of a B$^\varepsilon$-tree. When a node's buffer becomes full, messages are moved from that node's buffer to one of its children's buffers. The leaves of the B$^\varepsilon$-tree store key-value pairs, as in a B-tree. Point and range queries behave similarly to a B-tree, except that each buffer on the path from root to leaf must also be checked for items that affect the query.

B$^\varepsilon$-trees are asymptotically faster than B-trees, as summarized in Table 1. To see why, consider a B-tree with N items and in which each node can hold B keys. (For simplicity, assume keys have constant size and that the data associated with each key has negligiable size.) The tree will have fanout B, so its height will be $O(\log_B N)$. Inserts and lookups will therefore require $O(\log_B N)$ I/Os. A range query that returns k items will require $O(\log_B N + k/B)$ I/Os.

For comparison, a B$^\varepsilon$-tree with nodes of size B will have B^ε children, where $0 < \varepsilon \leq 1$. Each node will store one "pivot key" for each child, consuming B^ε space per node. The remaining $B - B^\varepsilon$ space in each node will be used to buffer newly inserted items. Since the fanout of the tree is B^ε, its height is $O(\log_{B^\varepsilon} N) = O(\frac{1}{\varepsilon}\log_B N)$. Consequently, searches will be slower by a factor of $\frac{1}{\varepsilon}$. However, each time a node flushes items to one of its children, it will move at least $(B - B^\varepsilon)/B^\varepsilon \approx B^{1-\varepsilon}$ elements. Since each element must be flushed $O(\frac{1}{\varepsilon}\log_B N)$ times to reach a leaf, the amortized cost of inserting N items is $O(\frac{1}{\varepsilon B^{1-\varepsilon}}\log_B N)$. Range queries returning k items require $O(\frac{1}{\varepsilon}\log_B N + k/B)$ I/Os. If we pick, for example, $\varepsilon = 1/2$, the point and range query costs of a B$^\varepsilon$-tree become $O(\log_B N)$ and $O(\log_B N + k/B)$, which are the same as a B-tree, but the insert cost becomes $O(\log_B N/\sqrt{B})$, which is faster by a factor of $\sqrt{B}$.

In practice, however, B$^\varepsilon$-trees use much larger nodes than B-trees. For example, a typical B-tree might use 4KB or 64KB nodes, compared to 4MB nodes in Tokutek's Fractal Tree indexes. B-trees must use small nodes because a node must be completely re-written each time a new item is added to the database, unlike in B$^\varepsilon$-trees, where writes are batched. Large nodes mean that, in practice, the height of a B$^\varepsilon$-tree is not much larger than the height of a B-tree on the same data. Thus, the performance of point queries in a B$^\varepsilon$-tree implementation can be comparable to point query performance in a B-tree. Large nodes also speed up range queries, since the data is spread over fewer nodes, requiring fewer disk seeks to read in all the data.

To get a feeling for what this speedup looks like, consider the following example. Suppose a key-value store holds 1TB of data, with 128-byte keys and records (key+value) of size 1KB. Suppose that data is logged for durability, and periodically all updates in the log are applied to the main tree in batch.

In the case of a B-tree with 4KB nodes, the fanout of the tree will be 4KB/128B$=32$. Thus the non-leaf nodes can comfortably fit into the memory of a typical server with 64GB of RAM, but only a negligible fraction of the 1TB of leaves will be cached at any given time. During a random insertion workload, most updates in a batch will require exactly 2 I/Os: 1 I/O to read in the target leaf and 1 I/O to write it back to disk after updating its contents.

For comparison, suppose a B$^\varepsilon$-tree has branching factor of 10 and nodes of size 1MB. Once again, all but the last level fit in memory. When a batch of logged updates is applied to the tree, they are simply stored in the tree's root buffer. Since the root is cached, this requires a single I/O. When an internal node becomes full and flushes its buffer to a non-leaf child, this causes two writes: an update of the parent and an update of the child. There are no reads required since both nodes are cached. When an internal node flushes its buffer to a leaf node, this requires one additional read to load the leaf into memory.

There are 1TB/1MB=2^{20} leaves, so since the tree has fanout 10, its height is $1 + \log_{10} 2^{20} \approx 7$. Each item is therefore involved in 14 I/Os: it is written and read once at each level.

However, each flush moves 1MB/10 = 100kB of data, in other words, 100 items. Thus, the average per-item cost of flushing an item to a leaf is $14/100$. Since a B-tree would require 2 I/Os for each item, the B$^\varepsilon$-tree is able to insert data $2/(14/100) = 14.3$ times faster than a B-tree. As key-value pairs get smaller, say for metadata updates, this speedup factor grows.

In both cases, a point query requires a single I/O to read the corresponding leaf for the queried key. Range queries can be much faster, as the B$^\varepsilon$-tree seeks only once every 1MB vs once every 4KB in the B-tree.

Because buffered messages are variable length in our implementation, even with fixed-size nodes, B is not constant. Rather than fix ε, our implementation bounds the range of pivots per node (B^ε) between 4 and 16.

Upserts. B$^\varepsilon$-trees support "upserts," an efficient method for updating a key-value pair in the tree. When an application wants to update the value associated with key k in the B$^\varepsilon$-tree, it inserts a message $(k, (f, \Delta))$ into the tree, where f specifies a call-back function that can be used to apply the change specified by Δ to the old value associated with key k. This message is inserted into the tree like any other piece of data. However, every time the message is flushed from one node to a child C, the B$^{\varepsilon}$-tree checks whether C's buffer contains the old value v associated with key k. If it does, then it replaces v with $f(v,\Delta)$ and discards the upsert message from the tree. If an application queries for k before the callback function is applied, then the B$^{\varepsilon}$-tree computes $f(v,\Delta)$ on the fly while answering the query. This query is efficient because an upsert for key k always lies on the path from the root of the B$^{\varepsilon}$-tree to the leaf containing k. Thus, the upsert mechanism can speed up updates by one to two orders of magnitude without slowing down queries.

Log-structured merge trees. Log-structured merge trees (LSM trees) [23] are a WOI with many variants [28]. They typically consist of a logarithmic number of indexes (e.g., B-trees) of exponentially increasing size. Once an index at one level fills up, it is emptied by merging it into the index at the next larger level.

LSM trees can be tuned to have the same insertion complexity as a B$^{\varepsilon}$-tree, but queries in a naïvely implemented LSM tree can be slow, as shown in Table 1. Implementers have developed methods to improve the query performance, most notably using Bloom filters [6] for each B-tree. A point query for an element in the data structure is typically reported as improving to $O(\log_B N)$, thus matching a B-tree. Bloom filters are used in most LSM tree implementations (e.g., [3, 12, 17, 20]).

Bloom filters do not help in range queries, since the successor of any key may be in any level. In addition, the utility of Bloom filters degrades with the use of upserts, and upserts are key to the performance of B*e*trFS. To see why, note that in order to compute the result of a query, all relevant upserts must be applied to the key-value pair. If there are many upserts "in flight" at different levels of the LSM tree, then searches will need to be performed on each such level. Bloom filters can be helpful to direct us to the levels of interest, but that does not obviate the need for many searches, and since the leaves of the different LSM tree B-trees might require fetching, the search performance can degrade.

B*e*trFS uses B$^{\varepsilon}$-trees, as implemented in Tokutek's Fractal Tree indexes, because Fractal Tree indexes are the only WOI implementation that matches the query performance of B-trees for all workloads, including the upsert-intensive workloads generated by B*e*trFS. In short, LSMs match B-tree query times in special cases, and B$^{\varepsilon}$-trees match B-tree query times in general.

3 B*e*trFS Design

B*e*trFS is an in-kernel file system designed to take full advantage of the performance strengths of B$^{\varepsilon}$-trees. The overall system architecture is illustrated in Figure 1.

Data	Insert	Point Query		Range Query
Struct.		no Upserts	w/ Upserts	
B-tree	$\log_B N$	$\log_B N$	$\log_B N$	$\log_B N + \frac{k}{B}$
LSM	$\frac{\log_B N}{\varepsilon B^{1-\varepsilon}}$	$\frac{\log_B^2 N}{\varepsilon}$	$\frac{\log_B^2 N}{\varepsilon}$	$\frac{\log_B^2 N}{\varepsilon} + \frac{k}{B}$
LSM+BF	$\frac{\log_B N}{\varepsilon B^{1-\varepsilon}}$	$\log_B N$	$\frac{\log_B^2 N}{\varepsilon}$	$\frac{\log_B^2 N}{\varepsilon} + \frac{k}{B}$
B$^{\varepsilon}$-tree	$\frac{\log_B N}{\varepsilon B^{1-\varepsilon}}$	$\frac{\log_B N}{\varepsilon}$	$\frac{\log_B N}{\varepsilon}$	$\frac{\log_B N}{\varepsilon} + \frac{k}{B}$

Table 1: Asymptotic I/O costs of important operations in B-trees and several different WOIs. Fractal Tree indexes simultaneously support efficient inserts, point queries (even in the presence of upserts), and range queries.

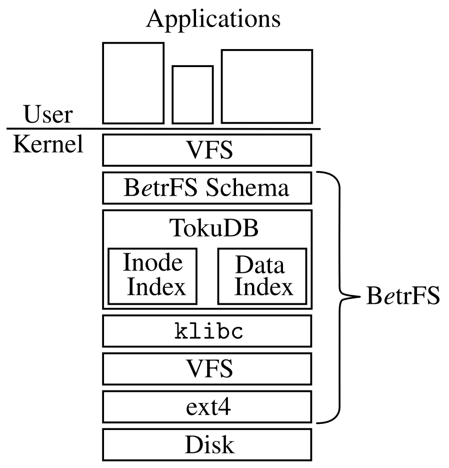


Figure 1: The B*e*trFS architecture.

The B*e*trFS VFS schema transforms file-system operations into efficient B$^{\varepsilon}$-tree operations whenever possible. The keys to obtaining good performance from B$^{\varepsilon}$-trees are (1) to use upsert operations to update file system state and (2) to organize data so that file-system scans can be implemented as range queries in the B$^{\varepsilon}$-trees. We describe how our schema achieves these goals in Section 4.

By implementing B*e*trFS as an in-kernel file system, we avoid the performance overheads of FUSE, which can be particularly deleterious for a write-optimized file system. We also expose opportunities for optimizing our file system's interaction with the kernel's page cache.

B*e*trFS's stacked file-system design cleanly separates the complex task of implementing write-optimized indexes from block allocation and free-space management. Our kernel port of TokuDB stores data on an underlying `ext4` file system, but any file system should suffice.

Porting a 45KLoC database into the kernel is a nontrivial task. We ported TokuDB into the kernel by writing a shim layer, which we call `klibc`, that translates the TokuDB external dependencies into kernel functions for locking, memory allocation, and file I/O. Section 6

```
/home
/home/alice
/home/bob
/home/alice/betrfs.pdf
/home/alice/betrfs.tex
/home/bob/betrfs.c
/home/bob/betrfs.ko
```

Figure 2: Example of the sort order used in B*e*trFS.

describes `klibc` and summarizes our experiences and lessons learned from the project.

4 The B*e*trFS File-System Schema

B$^{\varepsilon}$-trees implement a key-value store, so B*e*trFS must translate file-system operations into key-value operations. This section presents the B*e*trFS schema for performing this translation and explains how this schema takes advantage of the performance strengths of B$^{\varepsilon}$-trees.

4.1 B*e*trFS Data and Metadata Indexes

B*e*trFS stores file system data and metadata using two indexes in the underlying database: a metadata index and a data index. Since both keys and values may be variable-sized, B*e*trFS is able to pack many index entries into each B$^{\varepsilon}$-tree node.

The metadata index. The B*e*trFS prototype maintains an index mapping full pathnames (relative to the mount point) to file metadata (roughly equivalent to the contents of `struct stat`):

$$\text{path} \rightarrow (\text{size}, \text{owner}, \text{timestamps}, \text{etc}\ldots)$$

The metadata index is designed to support efficient file creations, deletions, lookups, and directory scans. The index sorts paths first by the number of slashes, then lexicographically. Thus, items within the same directory are stored consecutively as illustrated in Figure 2. With this ordering, scanning a directory, either recursively or not, can be implemented as a range query.

The data index. Though keys and values may be variable-sized, the B*e*trFS prototype breaks files into 4096-byte blocks for better integration with the page cache. Thus, the data index maps (file, offset) tuples to blocks:

$$(\text{path}, \text{block-number}) \rightarrow \text{data}[4096]$$

Keys in the data index are also sorted lexicographically, which guarantees that the contents of a file are logically adjacent and therefore, as explained in Subsection 2.2, almost always physically adjacent on disk. This enables file contents to be read sequentially at near disk bandwidth. B*e*trFS implements sparse files by simply omitting the sparse blocks from the data index.

B*e*trFS uses variable-sized values to avoid zero-padding the last block of each file. This optimization avoids the CPU overhead of zeroing out unused regions of a buffer, and then compressing the zeros away before writing a node to disk. For small-file benchmarks, this optimization yielded a significant reduction in overheads. For instance, this optimization improves throughput on TokuBench (§7) by 50–70%.

FS Operation	B$^{\varepsilon}$-tree Operation
Mkdir	Upsert
Rmdir	Upsert
Create	Upsert
Unlink	Upsert + Delete data blocks
Truncate	Upsert + Delete data blocks
Setattr (e.g. `chmod`)	Upsert
Rename	Copy files
Symlink	Upsert
Lookup (i.e. `lstat`)	Point Query
Readlink	Point Query
Readdir	Range Query
File write	Upsert
File read	Range Query
MMap readpage(s)	Point/Range Query
MMap writepage(s)	Upsert(s)

Table 2: B*e*trFS implementation strategies for basic file-system operations. Almost all operations are implemented using efficient upserts, point queries, or range queries. Unlink, Truncate, and Rename currently scale with file and/or directory sizes.

4.2 Implementing B*e*trFS Operations

Favoring blind writes. A latent assumption in much file system code is that data must be written at disk-sector granularity. As a result, a small write must first bring the surrounding disk block into the cache, modify the block, and then write it back. This pattern is reflected in the Linux page cache helper function `__block_write_begin()`. B*e*trFS avoids this read-modify-write pattern, instead issuing blind writes—writes without reads—whenever possible.

Reading and writing files in B*e*trFS. B*e*trFS implements file reads using range queries in the data index. B$^{\varepsilon}$-trees can load the results of a large range query from disk at effectively disk bandwidth.

B*e*trFS supports efficient file writes of any size via upserts and inserts. Application writes smaller than one 4K block become messages of the form:

$$\text{UPSERT}(\text{WRITE}, (\mathit{path}, n), \mathit{offset}, v, \ell),$$

which means the application wrote ℓ bytes of data, v, at the given *offset* into block *n* of the file specified by *path*. Upsert messages completely encapsulate a block modification, obviating the need for read-modify-write. Writes of an entire block are implemented with an insert (also called a put), which is a blind replacement of the block, and behaves similarly.

As explained in Subsection 2.2, upserts and inserts are messages inserted into the root node, which percolate down the tree. By using upserts for file writes, a write-optimized file system can aggregate many small random writes into a single large write to disk. Thus, the data can be committed to disk by performing a single seek and one large write, yielding an order-of-magnitude boost in performance for small random writes.

In the case of large writes spanning multiple blocks, inserts follow a similar path of copying data from the root to the leaves. The B$^{\varepsilon}$-tree implementation has some optimizations for large writes to skip intermediate nodes on the path to a leaf, but they are not aggressive enough to achieve full disk bandwidth for large sequential file writes. We leave this issue for future work, as a solution must carefully address several subtle issues related to pending messages and splitting leaf nodes.

File-system metadata operations in B*e*trFS. As shown in Table 2, B*e*trFS also converts almost all metadata updates, such timestamp changes, file creation and symbolic linking, into upserts.

The only metadata updates that are not upserts in B*e*trFS are unlink, truncate, and rename. We now explain the obstacles to implementing these operations as upserts, which we leave for future work.

Unlink and truncate can both remove blocks from a file. B*e*trFS performs this operation in the simplest possible way: it performs a range query on the blocks that are to be deleted, and issues a TokuDB delete for each such block in the data index. Although TokuDB delete operations are implemented using upserts, issuing $O(n)$ upserts can make this an expensive task.

Second, keying by full path makes recursive directory traversals efficient, but makes it non-trivial to implement efficient renames. For example, our current implementation renames files and directories by re-inserting all the key-value pairs under their new keys and then deleting the old keys, effectively performing a deep copy of the file or directory being renamed. One simple solution is to add an inode-style layer of indirection, with a third index. This approach is well-understood, and can sacrifice some read locality as the tree ages. We believe that data-structure-level optimizations can improve the performance of rename, which we leave for future work.

Although the schema described above can use upserts to make most changes to the file system, many POSIX file system functions specify preconditions that the OS must check before changing the file system. For example, when creating a file, POSIX requires the OS to check that the file doesn't already exist and that the user has write permission on the containing directory. In our experiments, the OS cache of file and directory information was able to answer these queries, enabling file creation etc., to run at the full speed of upserts.

Crash consistency. We use the TokuDB transaction mechanism for crash consistency. TokuDB transactions are equivalent to full data journaling, with all data and metadata updates logged to a file in the underlying `ext4` file system. Log entries are retired in-order, and no updates are applied to the tree on disk ahead of the TokuDB logging mechanism. Entries are appended to one of two in-memory log buffers (16 MB by default). These buffers are rotated and flushed to disk every second or when a buffer overflows.

Although exposing a transactional API may be possible, B*e*trFS currently uses transactions only as an internal consistency mechanism. B*e*trFS generally uses a single transaction per system call, except for writing data, which uses a transaction per data block. In our current implementation, transactions on metadata execute while holding appropriate VFS-level mutex locks, making transaction conflicts and aborts vanishingly rare.

Compression. Compression is important to performance, especially for keys. Both indexes use full paths as keys, which can be long and repetitive, but TokuDB's compression mitigates these overheads. Using `quicklz` [24], the sorted path names in our experiments compress by a factor of 20, making the disk-space overhead manageable.

The use of data compression also means that there isn't a one-to-one correspondence between reading a file-system-level block and reading a block from disk. A leaf node is typically 4 MB, and compression can pack more than 64 file system blocks into a leaf. In our experience with large data reads and writes, data compression can yield a boost to file system throughput, up to 20% over disabling compression.

5 Write-Optimization in System Design

In designing B*e*trFS, we set the goal of working within the existing Linux VFS framework. An underlying challenge is that, at points, the supporting code assumes that reads are as expensive as writes, and necessary for update-in-place. The use of write optimization violates these assumptions, as sub-block writes can be faster than a read. This section explains several strategies we found for improving the B*e*trFS performance while retaining Linux's supporting infrastructure.

5.1 Eternal Sunshine of the Spotless Cache

B*ε*trFS leverages the Linux page cache to implement efficient small reads, avoid disk reads in general, and facilitate memory-mapped files. By default, when an application writes to a page that is currently in cache, Linux marks it as dirty and writes it out later. This way, several application-level writes to a page can be absorbed in the cache, requiring only a single write to disk. In B*ε*trFS, however, small writes are so cheap that this optimization does not always make sense.

In B*ε*trFS, the `write` system call never dirties a clean page in the cache. When an application writes to a clean cached page, B*ε*trFS issues an upsert to the B^{ε}-tree and applies the write to the cached copy of that page. Thus the contents of the cache are still in sync with the on-disk data, and the cached page remains clean.

Note that B*ε*trFS' approach is not always better than absorbing writes in the cache and writing back the entire block. For example, if an application performs hundreds of small writes to the same block, then it would be more efficient to mark the page dirty and wait until the application is done to write the final contents back to disk. A production version of B*ε*trFS should include heuristics to detect this case. We found that performance in our prototype was good enough without this optimization, though, so we have not yet implemented it.

The only situation where a page in the cache is dirtied is when the file is memory-mapped for writing. The memory management hardware does not support fine-grained tracking of writes to memory-mapped files — the OS knows only that something within in the page of memory has been modified. Therefore, B*ε*trFS' `mmap` implementation uses the default read and write page mechanisms, which operate at page granularity.

Our design keeps the page cache coherent with disk. We leverage the page cacahe for faster warm-cache reads, but avoid unnecessary full-page writebacks.

5.2 FUSE is Write De-Optimized

We implemented B*ε*trFS as an in-kernel file system because the FUSE architecture contains several design decisions that can ruin the potential performance benefits of a write-optimized file system. FUSE has well-known overheads from the additional context switches and data marshalling it performs when communicating with user-space file systems. However, FUSE is particularly damaging to write-optimized file systems for completely different reasons.

FUSE can transform write-intensive into read-intensive workloads because it issues queries to the user-space file system before (and, in fact, after) most file system updates. For example, FUSE issues `GETATTR` calls (analogous to calling `stat()`) for the entire path of a file lookup, every time the file is looked up by an application. For most in-kernel file systems, subsequent lookups could be handled by the kernel's directory cache, but FUSE conservatively assumes that the underlying file system can change asynchronously (which can be true, e.g. in network file systems).

These searches can choke a write-optimized data structure, where insertions are two orders of magnitude faster than searches. The TokuFS authors explicitly cite these searches as the cause of the disappointing performance of their FUSE implementation [15].

The TableFS authors identified another source of FUSE overhead: double caching of inode information in the kernel [25]. This reduces the cache's effective hit rate. For slow file systems, the overhead of a few extra cache misses may not be significant. For a write-optimized data structure working on a write-intensive workload, the overhead of the cache misses can be substantial.

5.3 Ext4 as a Block Manager

Since TokuDB stores data in compressed nodes, which can have variable size, TokuDB relies on an underlying file system to act as a block and free space manager for the disk. Conventional file systems do a good job of storing blocks of large files adjacently on disk, especially when writes to the file are performed in large chunks.

Rather than reinvent the wheel, we stick with this design in our kernel port of TokuDB. B*ε*trFS represents tree nodes as blocks within one or more large files on the underlying file system, which in our prototype is unmodified `ext4` with ordered data mode and direct IO. We rely on `ext4` to correctly issue barriers to the disk write cache, although disabling the disk's write cache did not significantly impact performance of our workloads. In other words, all B*ε*trFS file system updates, data or metadata, generally appear as data writes, and an `fsync` to the underlying `ext4` file system ensures durability of a B*ε*trFS log write. Although there is some duplicated work between the layers, we expect ordered journaling mode minimizes this, as a typical B*ε*trFS instance spans 11 files from `ext4`'s perspective. That said, these redundancies could be streamlined in future work.

6 Implementation

Rather than do an in-kernel implementation of a write-optimized data structure from scratch, we ported TokuDB into the Linux kernel as the most expedient way to obtain a write-optimized data structure implementation. Such data structure implementations can be com-

Component	Description	Lines
VFS Layer	Translate VFS hooks to TokuDB queries.	1,987
TokuDB	Kernel version of TokuDB. (960 lines changed)	44,293
`klibc`	Compatibility wrapper.	4,155
Linux	Modifications	58

Table 3: Lines of code in B*e*trFS, by component.

Class	ABIs	Description
Memory	4	Allocate buffer pages and heap objects.
Threads	24	Pthreads, condition variables, and mutexes.
Files	39	Access database backend files on underlying, disconnected file system.
zlib	7	Wrapper for kernel zlib.
Misc	27	Print errors, qsort, get time, etc..
Total	101	

Table 4: Classes of ABI functions exported by `klibc`.

plex, especially in tuning the I/O and asymptotic merging behavior.

In this section, we explain how we ported a large portion of the TokuDB code into the kernel, challenges we faced in the process, lessons learned from the experience, and future work for the implementation. Table 3 summarizes the lines of code in B*e*trFS, including the code interfacing the VFS layer to TokuDB, the `klibc` code, and minor changes to the Linux kernel code, explained below. The B*e*trFS prototype uses Linux version 3.11.10.

6.1 Porting Approach

We initially decided the porting was feasible because TokuDB has very few library requirements and is written in a C-like form of C++. In other words, the C++ features used by TokuDB are primarily implemented at compile time (e.g., name mangling and better type checking), and did not require runtime support for features like exceptions. Our approach should apply to other WOIs, such as an LSM tree, inasmuch as the implementation follows a similar coding style.

As a result, we were able to largely treat the TokuDB code we used as a binary blob, creating a kernel module (.ko file) from the code. We exported interfaces used by the B*e*trFS VFS layer to use C linkage, and similarly declared interfaces that TokuDB imported from `klibc` to be C linkage.

We generally minimized changes to the TokuDB code, and selected imported code at object-file granularity. In a few cases, we added compile-time macros to eliminate code paths or functions that would not be used yet required cumbersome dependencies. Finally, when a particularly cumbersome user-level API, such as `fork`, is used in only a few places, we rewrote the code to use a more suitable kernel API. We call the resulting set of dependencies imported by TokuDB `klibc`.

6.2 The `klibc` Framework

Table 4 summarizes the ABIs exported by `klibc`. In many cases, kernel ABIs were exported directly, such as `memcpy`, or were straightforward wrappers for features such as synchronization and memory allocation. In a few cases, the changes were more complex.

The use of `errno` in the TokuDB code presented a particular challenge. Linux passes error codes as negative return values, whereas `libc` simply returns negative one and places the error code in a per-thread variable `errno`. Checks for a negative value and reads of `errno` in TokuDB were so ubiquitous that changing the error-handling behavior was impractical. We ultimately added an `errno` field to the Linux `task` struct; a production implementation would instead rework the error-passing code.

Although wrapping pthread abstractions in kernel abstractions was fairly straightforward, static initialization and direct access to pthread structures created problems. The primary issue is that converting pthread abstractions to kernel abstractions replaced members in the pthread structure definitions. Static initialization would not properly initialize the modified structure. Once the size of pthread structures changed, we had to eliminate any code that imported system pthread headers, lest embedded instances of these structures calculate the wrong size.

In reusing `ext4` as a block store, we faced some challenges in creating module-level file handles and paths. File handles were more straightforward: we were able to create a module-level handle table and use the pread (cursor-less) API to `ext4` for reads and writes. We did have to modify Linux to export several VFS helper function that accepted a `struct file` directly, rather than walking the process-level file descriptor table. We also modified `ext4` to accept input for reads with the `O_DIRECT` flag that were not from a user-level address.

When B*e*trFS allocates, opens, or deletes a block store on the underlying `ext4` file system, the module essentially `chroots` into an `ext4` file system disconnected from the main tree. Because this is kernel code, we also wish to avoid permission checks based on the current process's credentials. Thus, path operations include a "context switch" operation, where the current task's file system root and credentials are saved and restored.

6.3 Changes to TokuDB

With a few exceptions, we were able to use TokuDB in the kernel without major modifications. This subsection outlines the issues that required refactoring the code.

The first issue we encountered was that TokuDB makes liberal use of stack allocation throughout. One function allocated a 12KB buffer on the stack! In contrast, stack sizes in the Linux kernel are fixed at compile time, and default to 8KB. In most cases, we were able to use compile-time warnings to identify large stack allocation and convert them to heap allocations and add free functions. In the case where these structures were performance-critical, such as a database cursor, we modified the TokuDB code to use faster allocation methods, such as a kernel cache or per-CPU variable. Similarly, we rewrote several recursive functions to use a loop. Nonetheless, we found that deep stacks of more modest-sized frames were still possible, and increased the stack size to 16 KB. We plan to reign in the maximum stack size in future work.

Finally, we found a small mismatch between the behavior of futexes and kernel wait queues that required code changes. Essentially, recent implementations of pthread condition variables will not wake a sleeping thread up due to an irrelevant interrupt, making it safe (though perhaps inadvisable) in user space not to double-check invariants after returning from `pthread_cond_wait`. The Linux-internal equivalents, such as `wait_event`, can spuriously wake up a thread in a way that is difficult to distinguish without re-checking the invariant. Thus, we had to place all `pthread_cond_wait` calls in a loop.

6.4 Future Work and Limitations

The B*e*trFS prototype is an ongoing effort. The effort has reached sufficient maturity to demonstrate the power of write optimization in a kernel file system. However, there are several points for improvement in future work.

The most useful feature currently missing from the TokuDB codebase is range upserts; upserts can only be applied to a single key, or broadcast to all keys. Currently, file deletion must be implemented by creating a remove upsert for each data block in a file; the ability to create a single upsert applied to a limited range would be useful, and we leave this for future work. The primary difficulty in supporting such an abstraction is tuning how aggressively the upsert should be flushed down to the leaves versus applied to point queries on demand; we leave this issue for future work as well.

One subtle trade-off in organizing on-disk placement is between rename and search performance. B*e*trFS keys files by their path, which currently results in rename copying the file from one disk location to another. This can clearly be mitigated by adding a layer of indirection (i.e., an inode number); however, this is at odds with the goal of preserving data locality within a directory hierarchy. We plan to investigate techniques for more efficient directory manipulation that preserve locality. Similarly, our current prototype does not support hard links.

Our current prototype also includes some double caching of disk data. Nearly all of our experiments measure cold-cache behavior, so this does not affect the fidelity of our results. Profligate memory usage is nonetheless problematic. In the long run, we intend to better integrate these layers, as well as eliminate emulated file handles and paths.

7 Evaluation

We organize our evaluation around the following questions:

- Are microwrites on B*e*trFS more efficient than on other general-purpose file systems?
- Are large reads and writes on B*e*trFS at least competitive with other general-purpose file systems?
- How do other file system operations perform on B*e*trFS?
- What are the space (memory and disk) overheads of B*e*trFS?
- Do applications realize better overall performance on B*e*trFS?

Unless otherwise noted, benchmarks are cold-cache tests. All file systems benefit equally from hits in the page and directory caches; we are interested in measuring the efficiency of cache misses.

All experimental results were collected on a Dell Optiplex 790 with a 4-core 3.40 GHz Intel Core i7 CPU, 4 GB RAM, and a 250 GB, 7200 RPM ATA disk. Each file system used a 4096-byte block size. The system ran Ubuntu 13.10, 64-bit, with Linux kernel version 3.11.10. Each experiment compared with several general purpose file systems, including `btrfs`, `ext4`, `XFS`, and `zfs`. Error bars and $\pm$ ranges denote 95% confidence intervals.

7.1 Microwrites

We evaluated microwrite performance using both metadata and data intensive microbenchmarks. To exercise file creation, we used the TokuBench benchmark [15] to create 500,000 200-byte files in a balanced directory tree with a fanout of 128. The results are shown in Figure 3. TokuBench also measures the scalability of the file system as threads are added; we measured up to 4 threads since our machine has 4 cores.

B*e*trFS exhibited substantially higher throughput than the other file systems. The closest competitor was `zfs` at 1 thread; as more threads were added, the gap widened considerably. Compared to `ext4`, `XFS`, and `btrfs`, B*e*trFS throughput was an order of magnitude higher.

This performance distinction is attributable to both fewer total writes and fewer seeks per byte written—i.e., better aggregation of small writes. Based on profiling from `blktrace`, one major distinction is total bytes written: B*e*trFS writes 4–10$\times$ fewer total MB to disk, with

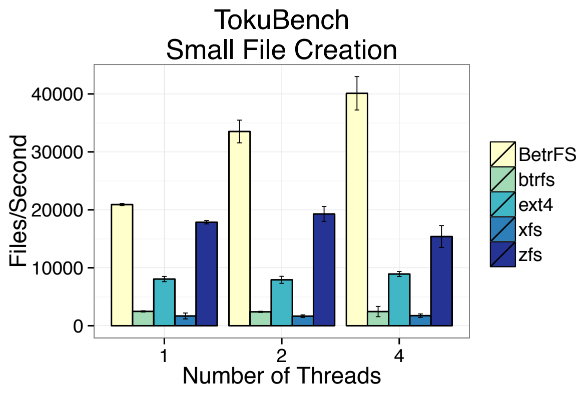


Figure 3: Total time to create 500,000 200-byte files, using 1, 2, and 4 threads. We measure the number of files created per second. Higher is better.

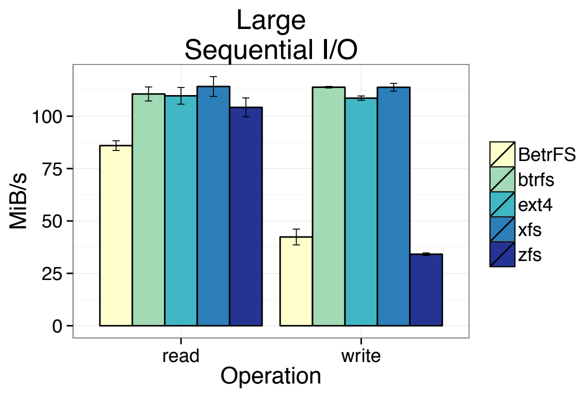


Figure 4: Large file I/O. We sequentially read and write 1GiB files. Higher is better.

File System	Time (s)
B*e*trFS	0.17 ± 0.01
`ext4`	11.60 ± 0.39
`XFS`	11.71 ± 0.28
`btrfs`	11.60 ± 0.38
`zfs`	14.75 ± 1.45

Table 5: Time in seconds to execute 1000 4-byte microwrites within a 1GiB file. Lower is better.

an order of magnitude fewer total write requests. Among the other file systems, `ext4`, `XFS`, and `zfs` wrote roughly the same amount of data, but realized widely varying underlying write throughput. The only file system with a comparable write throughput was `zfs`, but it wrote twice as much data using 12.7× as many disk requests.

To measure microwrites to files, we wrote a custom benchmark that performs 1,000 random 4-byte writes within a 1GiB file, followed by an `fsync()`. Table 5 lists the results. B*e*trFS was two orders of magnitude faster than the other file systems.

These results demonstrate that B*e*trFS improves microwrite performance by one to two orders of magnitude compared to current general-purpose file systems.

7.2 Large Reads and Writes

We measured the throughput of sequentially reading and writing a 1GiB file, 10 blocks at a time. We created the file using random data to avoid unfairly advantaging compression in B*e*trFS. In this experiment, B*e*trFS benefits from compressing keys, but not data. We note that with compression and moderately compressible data, B*e*trFS can easily exceed disk bandwidth. The results are illustrated in Figure 4.

In general, most general-purpose file systems can read and write at disk bandwidth. In the case of a large sequential reads, B*e*trFS can read data at roughly 85 MiB/s. This read rate is commensurate with overall disk utilization, which we believe is a result of less aggressive read-ahead than the other file systems. We believe this can be addressed by re-tuning the TokuDB block cache prefetching behavior.

In the case of large writes, the current B*e*trFS prototype achieved just below half of the disk's throughput. The reason for this is that each block write must percolate down the interior tree buffers; a more efficient heuristic would detect a large streaming write and write directly to a leaf. As an experiment, we manually forced writes to the leaf in an empty tree, and found write throughput comparable to the other file systems. That said, applying this optimization is somewhat tricky, as there are a number of edge cases where leaves must be read and rewritten or messages must be flushed. We leave this issue for future work.

7.3 Directory Operations

In this section, we measure the impact of the B*e*trFS design on large directory operations. Table 6 reports the time taken to run `find`, `grep -r`, `mv`, and `rm -r` on the Linux 3.11.10 source tree, starting from a cold cache. The `grep` test recursively searches the file contents for the string "cpu_to_be64", and the `find` test searches for files named "wait.c". The `rename` test renames the entire kernel source tree, and the `delete` test does a recursive removal of the source.

Both the `find` and `grep` benchmarks demonstrate the value of sorting files and their metadata lexicographically by full path, so that related files are stored near each other on disk. B*e*trFS can search directory metadata and file data one or two orders of magnitude faster than other file systems, with the exception of `grep` on `btrfs`, which is

FS	find	grep	dir rename	delete
B*e*trFS	0.36 ± 0.06	3.95 ± 0.28	21.17 ± 1.01	46.14 ± 1.12
btrfs	14.91 ± 1.18	3.87 ± 0.94	0.08 ± 0.05	7.82 ± 0.59
ext4	2.47 ± 0.07	46.73 ± 3.86	0.10 ± 0.02	3.01 ± 0.30
XFS	19.07 ± 3.38	66.20 ± 15.99	19.78 ± 5.29	19.78 ± 5.29
zfs	11.60 ± 0.81	41.74 ± 0.64	14.73 ± 1.64	14.73 ± 1.64

Table 6: Directory operation benchmarks, measured in seconds. Lower is better.

FS	chmod	mkdir	open	read	stat	unlink	write
B*e*trFS	4913 ± 0.27	67072 ± 25.68	1697 ± 0.12	561 ± 0.01	1076 ± 0.01	47873 ± 7.7	32142 ± 4.35
btrfs	4574 ± 0.27	24805 ± 13.92	1812 ± 0.12	561 ± 0.01	1258 ± 0.01	26131 ± 0.73	3891 ± 0.08
ext4	4970 ± 0.14	41478 ± 18.99	1886 ± 0.13	556 ± 0.01	1167 ± 0.05	16209 ± 0.2	3359 ± 0.04
XFS	5342 ± 0.21	73782 ± 19.27	1757 ± 0.12	1384 ± 0.07	1134 ± 0.02	19124 ± 0.32	9192 ± 0.28
zfs	36449 ± 118.37	171080 ± 307.73	2681 ± 0.08	6467 ± 0.06	1913 ± 0.04	78946 ± 7.37	18382 ± 0.42

Table 7: Average time in cycles to execute a range of common file system calls. Lower is better.

comparable.

Both the rename and delete tests show the worst-case behavior of B*e*trFS. Because B*e*trFS does not include a layer of indirection from pathname to data, renaming requires copying all data and metadata to new points in the tree. We also measured large-file renames, and saw similarly large overheads—a function of the number of blocks in the file. Although there are known solutions to this problem, such as by adding a layer of indirection, we plan to investigate techniques that can preserve the appealing lexicographic locality without sacrificing rename and delete performance.

7.4 System Call Nanobenchmarks

Finally, Table 7 shows timings for a nanobenchmark that measures various system call times. Because this nanobenchmark is warm-cache, it primarily exercises the VFS layer. B*e*trFS is close to being the fastest file system on `open`, `read`, and `stat`. On `chmod`, `mkdir`, and `unlink`, B*e*trFS is in the middle of the pack.

Our current implementation of the `write` system call appears to be slow in this benchmark because, as mentioned in Section 5.1, writes in B*e*trFS issue an upsert to the database, even if the page being written is in cache. This can be advantageous when a page is not written often, but that is not the case in this benchmark.

7.5 Space Overheads

The Fractal Tree index implementation in B*e*trFS includes a cachetable, which caches tree nodes. Cachetable memory is bounded. B*e*trFS triggers background flushing when memory exceeds a low watermark and forces writeback at a high watermark. The high watermark is currently set to one eighth of total system memory. This is configurable, but we found that additional cachetable memory had little performance impact in our workloads.

	Total B*e*trFS Disk Usage (GiB)		
Input Data	After Writes	After Deletes	After Flushes
4	4.14 ± 0.07	4.12 ± 0.00	4.03 ± 0.12
16	16.24 ± 0.06	16.20 ± 0.00	10.14 ± 0.21
32	32.33 ± 0.02	32.34 ± 0.00	16.22 ± 0.00
64	64.57 ± 0.06	64.59 ± 0.00	34.36 ± 0.18

Table 8: B*e*trFS disk usage, measured in GiB, after writing large incompressible files, deleting half of those files, and flushing B^{ε}-tree nodes.

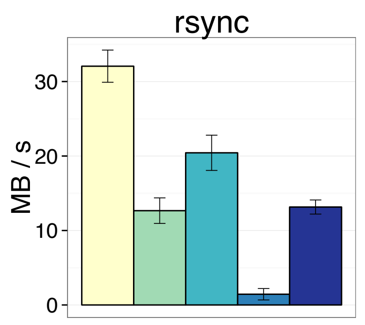
No single rule governs B*e*trFS disk usage, as stale data may remain in non-leaf nodes after delete, rename, and overwrite operations. Background cleaner threads attempt to flush pending data from 5 internal nodes per second. This creates fluctuation in B*e*trFS disk usage, but overheads swiftly decline at rest.

To evaluate the B*e*trFS disk footprint, we wrote several large incompressible files, deleted half of those files, and then initiated a B^{ε}-tree flush. After each operation, we calculated the B*e*trFS disk usage using `df` on the underlying `ext4` partition.

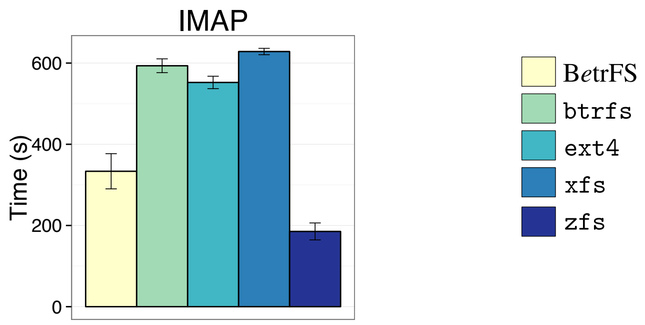
Writing new data to B*e*trFS introduced very little overhead, as seen in Table 8. For deletes, however, B*e*trFS issues an upsert for every file block, which had little impact on the B*e*trFS footprint because stale data is lazily reclaimed. After flushing, there was less than 3GiB of disk overhead, regardless of the amount of live data.

7.6 Application Performance

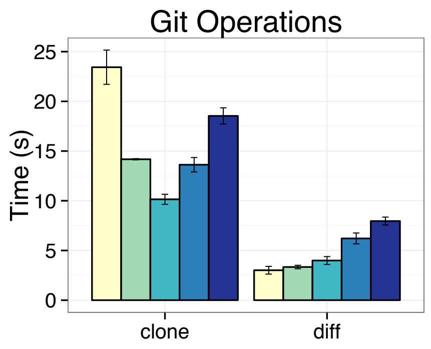
Figure 5 presents performance measurements for a variety of metadata-intensive applications. We measured the time to `rsync` the Linux 3.11.10 source code to a new di-



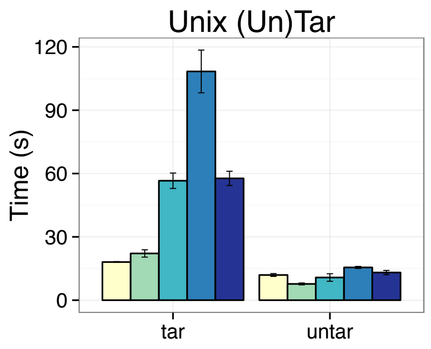
(a) `rsync` of Linux 3.11.10 source. The data source and destination are within the same partition and file system. Throughput in MB/s, higher is better.



(b) IMAP benchmark, 50% message reads, 50% marking and moving messages among inboxes. Execution time in seconds, lower is better.



(c) Git operations. The BetrFS source repository was `git-cloned` locally, and a `git-diff` was taken between two milestone commits. Lower is better.



(d) Unix `tar` operations. The Linux version 3.11.10 source code was both `tar`ed and un-`tar`ed. Time is in seconds. Lower is better.

Figure 5: Application benchmarks

rectory on the same file system, using the `--in-place` option to avoid temporary file creation (Figure 5a). We performed a benchmark using version 2.2.13 of the Dovecot mail server to simulate IMAP client behavior under a mix of read requests, requests to mark messages as read, and requests to move a message to one of 10 other folders. The balance of requests was 50% reads, 50% flags or moves. We exercised the git version control system using a `git-clone` of the local BetrFS source tree and a `git-diff` between two milestone commits (Figure 5c). Finally, we measured the time to `tar` and un-`tar` the Linux 3.11.10 source (Figure 5d).

BetrFS yielded substantially higher performance on several applications, primarily applications with microwrites or large streaming reads or writes. In the case of the IMAP benchmark, marking or moving messages is a small-file rename in a large directory—a case BetrFS handled particularly well, cutting execution time in half compared to most other file systems. Note that the IMAP test is a sync-heavy workload, issuing over 26K `fsync()` calls over 334 seconds, each forcing a full BetrFS log flush. `rsync` on BetrFS realized significantly higher throughput because writing a large number of modestly sized files in lexicographic order is, on BetrFS, aggregated into large, streaming disk writes. Similarly, `tar` benefited from both improved reads of many files in lexicographic order, as well as efficient aggregation of writes. `tar` on BetrFS was only marginally better than `btrfs`, but the execution time was at least halved compared to `ext4` and `XFS`.

The only benchmark significantly worse on BetrFS was `git-clone`, which does an `lstat` on every new file before creating it—despite cloning into an empty directory. Here, a slower, small read obstructs a faster write. For comparison, the `rsync --in-place` test case illustrates that, if an application eschews querying for the existence of files before creating them, BetrFS can deliver substantial performance benefits.

These experiments demonstrate that several real-world, off-the-shelf applications can benefit from executing on a write-optimized file system *without application modifications.* With modest changes to favor blind writes, applications can perform even better. For most applications not suited to write optimization, performance is not harmed or could be tuned.

8 Related Work

Previous write-optimized file systems. TokuFS [15] is an in-application library file system, also built using B$^\varepsilon$-trees. TokuFS showed that a write-optimized file system can support efficient write-intensive and scan-intensive workloads. TokuFS had a FUSE-based version, but the authors explicitly omitted disappointing measurements using FUSE.

KVFS [31] is based on a transactional variation of an LSM-tree, called a VT-tree. Impressively, the performance of their transactional, FUSE-based file system was comparable to the performance of the in-kernel `ext4` file system, which does not support transactions. One performance highlight was on random-writes, where they outperformed `ext4` by a factor of 2. They also used stitching to perform well on sequential I/O in the presence of LSM-tree compaction.

TableFS [25] uses LevelDB to store file-system metadata. They showed substantial performance improvements on metadata-intensive workloads, sometimes up to an order of magnitude. They used `ext4` as an object store for large files, so sequential I/O performance was comparable to `ext4`. They also analyzed the FUSE overhead relative to a library implementation of their file system and found that FUSE could cause a 1000× increase in disk-read traffic (see Figure 9 in their paper).

If these designs were ported to the kernel, we expect that they would see some, but not all, of the performance benefits of B*ε*trFS. Because the asymptotic behavior is better for B$^\varepsilon$-trees than LSMs in some cases, we expect the performance of an LSM-based file system will not be completely comparable.

Other WOIs. COLAs [4] are an LSM tree variant that uses fractional cascading [13] to match the performance of B$^\varepsilon$-trees for both insertions and queries, but we are not aware of any full featured, production-quality COLA implementation. xDict [8] is a cache-oblivious WOI with asymptotic behavior similar to a B$^\varepsilon$-tree.

Key-Value Stores. WOIs are widely used in key-value stores, including BigTable [12], Cassandra [20], HBase [3], LevelDB [17], TokuDB [33] and TokuMX [34]. BigTable, Cassandra, and LevelDB use LSM-tree variants. TokuDB and TokuMX use Fractal Tree indexes. LOCS [35] optimizes LSM-trees for a key-value store on a multi-channel SSD.

Instead of using WOIs, FAWN [1] writes to a log and maintains an in-memory index for queries. SILT [22] further reduces the design's memory footprint during the merging phase.

Alternatives to update-in-place. The Write Anywhere File Layout (WAFL) uses files to store its metadata, giving it incredible flexibility in its block allocation and layout policies [19]. WAFL does not address the microwrite problem, however, as its main goal is to provide efficient copy-on-write snapshots.

Log-structured File Systems and their derivatives [1, 21, 22, 26] are write-optimized in the sense that they log data, and are thus very fast at ingesting file system changes. However, they still rely on read-modify-write for file updates and suffer from fragmentation.

Logical logging is a technique used by some databases in which operations, rather than the before and after images of individual database pages, are encoded and stored in the log [18]. Like a logical log entry, an upsert message encodes a mutation to a value in the key-value store. However, an upsert is a first-class storage object. Upsert messages reside in B$^\varepsilon$-tree buffers and are evaluated on the fly to satisfy queries, or to be merged into leaf nodes.

9 Conclusion

The B*ε*trFS prototype demonstrates that write-optimized indexes are a powerful tool for file-system developers. In some cases, B*ε*trFS out-performs traditional designs by orders of magnitude, advancing the state of the art over previous results. Nonetheless, there are some cases where additional work is needed, such as further data-structure optimizations for large streaming I/O and efficient renames of directories. Our results suggest that further integration and optimization work is likely to yield even better performance results.

Acknowledgments

We thank the engineers at Tokutek for developing and open-sourcing a wonderful B$^\varepsilon$-tree implementation from which we built B*ε*trFS. We thank the anonymous reviewers and our shepherd, Geoff Kuenning, for their insightful comments on earlier drafts of the work. Vrushali Kulkarni, Oleksii Starov, Sourabh Yerfule, and Ahmad Zaraei contributed to the B*ε*trFS prototype. This research was supported in part by NSF grants CNS-1409238, CNS-1408782, CNS-1408695, CNS-1405641, CNS-1149229, CNS-1161541, CNS-1228839, CNS-1408782, IIS-1247750, CCF-1314547 and the Office of the Vice President for Research at Stony Brook University.

References

[1] ANDERSEN, D. G., FRANKLIN, J., KAMINSKY, M., PHANISHAYEE, A., TAN, L., AND VASUDEVAN, V. Fawn: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), pp. 1–14.

[2] APACHE. Accumulo. `http://accumulo.apache.org`.

[3] APACHE. HBase. `http://hbase.apache.org`.

[4] BENDER, M. A., FARACH-COLTON, M., FINEMAN, J. T., FOGEL, Y. R., KUSZMAUL, B. C., AND NELSON, J. Cache-oblivious streaming B-trees. In *Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)* (2007), pp. 81–92.

[5] BENT, J., GIBSON, G. A., GRIDER, G., MCCLELLAND, B., NOWOCZYNSKI, P., NUNEZ, J., POLTE, M., AND WINGATE, M. PLFS: A checkpoint filesystem for parallel applications. In *Proceedings of the ACM/IEEE Conference on High Performance Computing (SC)* (2009).

[6] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM 13*, 7 (1970), 422–426.

[7] BONWICK, J. ZFS: the last word in file systems. `https://blogs.oracle.com/video/entry/zfs_the_last_word_in`, Sept. 14 2004.

[8] BRODAL, G. S., DEMAINE, E. D., FINEMAN, J. T., IACONO, J., LANGERMAN, S., AND MUNRO, J. I. Cache-oblivious dynamic dictionaries with update/query tradeoffs. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)* (2010), pp. 1448–1456.

[9] BRODAL, G. S., AND FAGERBERG, R. Lower bounds for external memory dictionaries. In *Proceedings of the 14th Annual ACM-SIAM symposium on Discrete Algorithms (ACM)* (2003), pp. 546–554.

[10] BUCHSBAUM, A. L., GOLDWASSER, M., VENKATASUBRAMANIAN, S., AND WESTBROOK, J. R. On external memory graph traversal. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)* (2000), pp. 859–860.

[11] CARD, R., TS'O, T., AND TWEEDIE, S. Design and implementation of the Second Extended Filesystem. In *Proceedings of the First Dutch International Symposium on Linux* (1994), pp. 1–6.

[12] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS) 26*, 2 (2008), 4.

[13] CHAZELLE, B., AND GUIBAS, L. J. Fractional cascading: I. A data structuring technique. *Algorithmica 1*, 1-4 (1986), 133–162.

[14] DOUTHITT, D. Instant 10-20% boost in disk performance: the "noatime" option. `http://administratosphere.wordpress.com/2011/07/29/instant-10-20-boost-in-disk-performance-the-noatime-option/`, July 2011.

[15] ESMET, J., BENDER, M. A., FARACH-COLTON, M., AND KUSZMAUL, B. C. The TokuFS streaming file system. In *Proceedings of the 4th USENIX Workshop on Hot Topics in Storage (HotStorage)* (June 2012).

[16] File system in userspace. `http://fuse.sourceforge.net/`.

[17] GOOGLE, INC. LevelDB: A fast and lightweight key/value database library by Google. `http://code.google.com/p/leveldb/`.

[18] GRAY, J., AND REUTER, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[19] HITZ, D., LAU, J., AND MALCOLM, M. File system design for an NFS file server appliance. Tech. rep., NetApp, 1994.

[20] LAKSHMAN, A., AND MALIK, P. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review 44*, 2 (2010), 35–40.

[21] LEE, C., SIM, D., HWANG, J., AND CHO, S. F2FS: A new file system for flash storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2015).

[22] LIM, H., FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), pp. 1–13.

[23] O'NEIL, P., CHENG, E., GAWLIC, D., AND O'NEIL, E. The log-structured merge-tree (LSM-tree). *Acta Informatica 33*, 4 (1996), 351–385.

[24] QUICKLZ. Fast compression library for c, c#, and java. `http://www.quicklz.com/`.

[25] REN, K., AND GIBSON, G. A. TABLEFS: Enhancing metadata efficiency in the local file system. In *USENIX Annual Technical Conference* (2013), pp. 145–156.

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

[27] SEARS, R., CALLAGHAN, M., AND BREWER, E. A. Rose: compressed, log-structured replication. *Proceedings of the VLDB Endowment 1*, 1 (2008), 526–537.

[28] SEARS, R., AND RAMAKRISHNAN, R. bLSM: a general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), ACM, pp. 217–228.

[29] SELTZER, M., BOSTIC, K., MCKUSICK, M. K., AND STAELIN, C. An implementation of a log-structured file system for UNIX. In *Proceedings of the USENIX Winter 1993 Conference Proceedings* (San Diego, CA, Jan. 1993), p. 3.

[30] SELTZER, M., SMITH, K. A., BALAKRISHNAN, H., CHANG, J., MCMAINS, S., AND PADMANABHAN, V. File system logging versus clustering: A performance comparison. In *Proceedings of the USENIX 1995 Technical Conference Proceedings* (New Orleans, LA, Jan. 1995), p. 21.

[31] SHETTY, P., SPILLANE, R. P., MALPANI, R., ANDREWS, B., SEYSTER, J., AND ZADOK, E. Building workload-independent storage with VT-trees. In *FAST* (2013), pp. 17–30.

[32] SWEENY, A., DOUCETTE, D., HU, W., ANDERSON, C., NISHIMOTO, M., AND PECK, G. Scalability in the XFS file system. In *Proceedings of the 1996 USENIX Technical Conference* (San Diego, CA, Jan. 1996), pp. 1–14.

[33] TOKUTEK, INC. TokuDB: MySQL Performance, MariaDB Performance. `http://www.tokutek.com/products/tokudb-for-mysql/`.

[34] TOKUTEK, INC. TokuMX—MongoDB Performance Engine. `http://www.tokutek.com/products/tokumx-for-mongodb/`.

[35] WANG, P., SUN, G., JIANG, S., OUYANG, J., LIN, S., ZHANG, C., AND CONG, J. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), EuroSys '14, pp. 16:1–16:14.

SDGen: Mimicking Datasets for Content Generation in Storage Benchmarks

Raúl Gracia-Tinedo
Universitat Rovira i Virgili (Spain)
raul.gracia@urv.cat

Danny Harnik, Dalit Naor, Dmitry Sotnikov
IBM Research-Haifa (Israel)
{dannyh, dalit, dmitrys}@il.ibm.com

Sivan Toledo, Aviad Zuck
Tel-Aviv University (Israel)
{stoledo, aviadzuc}@tau.ac.il

Abstract

Storage system benchmarks either use samples of proprietary data or synthesize artificial data in *simple ways* (such as using zeros or random data). However, many storage systems behave completely differently on such artificial data than they do on real-world data. This is the case with systems that include data reduction techniques, such as compression and/or deduplication.

To address this problem, we propose a benchmarking methodology called *mimicking* and apply it in the domain of data compression. Our methodology is based on *characterizing the properties of real data* that influence the performance of compressors. Then, we use these characterizations to generate new synthetic data that mimics the real one in many aspects of compression. Unlike current solutions that only address the *compression ratio* of data, mimicking is flexible enough to also emulate *compression times* and *data heterogeneity*. We show that these properties matter to the system's performance.

In our implementation, called *SDGen*, characterizations take at most 2.5KB per data chunk (e.g., 64KB) and can be used to efficiently share benchmarking data in a highly anonymized fashion; sharing it carries few or no privacy concerns. We evaluated our data generator's accuracy on compressibility and compression times using real-world datasets and multiple compressors (`lz4, zlib, bzip2` and `lzma`). As a proof-of-concept, we integrated *SDGen* as a content generation layer in two popular benchmarks (LinkBench and Impressions).

1 Introduction

Benchmarking is a fundamental building block for researchers and practitioners to measure the performance of storage systems in a reproducible manner [35]. Indeed, the research community has taken remarkable steps towards accurately emulating observed workloads into experimental assessments. A myriad of examples can be found in the literature, such as benchmarks for file systems [7, 31], cloud storage [13, 16, 22] or databases [9, 15], to name a few.

However, most storage benchmarks do not pay particular attention to the *contents* generated during their execution [35] (see examples in Table 1). For instance, Impressions [7] implements accurate statistical methods to model the structure of a file system, but the contents of files are by default zeros or statically generated by third-party applications. Another example is OLTPBench [15], which provides a rich suite of database workloads and access patterns; however, the payload of queries is filled with random data. Clearly these contents are not realistic. Thus, the following question arises: *does the content matter to the performance analysis of systems?* The answer is a definitive *yes* when data reduction is involved.

Data reduction and performance sensitivity: To improve performance and capacity, a variety of storage systems integrate data reduction techniques [11, 21, 23, 25, 27]. This can have two crucial effects on the performance of the storage system: (i) When data is highly compressible, the amount of bytes actually written to the storage diminishes and performance can improve dramatically. (ii) The running time of compression algorithms varies greatly for different data types (e.g. [18]) and hence can affect the overall throughput and latency of the system. As a result, the performance of many systems with data reduction techniques is extremely content-sensitive.

To illustrate this, we measured the transfer times of ZFS, a file system with built-in compression [27, 38]. We copied sequentially 1GB files filled with low (random) and high (zeros) compressible data. The results in Fig. 1 support our claim: *the transfer times of ZFS greatly vary depending on the file contents*. Thus, two executions of the same benchmark may report disparate performance results of ZFS, just depending on the data used.

Current solutions: One solution to benchmarking a content-sensitive storage system is to use real life datasets for executing the tests. However, this practice is limiting due to the burden of copying large amounts of data onto the testing system. Even more so, privacy con-

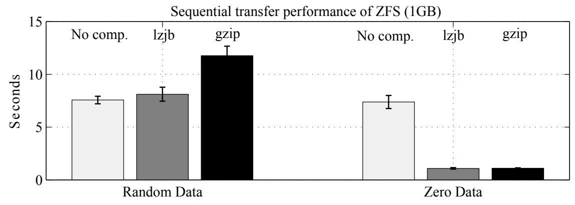


Figure 1: Sequential transfer times of ZFS depending on file contents with and without compression.

cerns greatly inhibit this approach as end users typically are unwilling to share their proprietary data [33].

Another practice which is gradually being adopted is generating synthetic data with definable compressibility. For example, *VDBench* [2], *Fio* [4] and *LinkBench* [9] all offer synthetic data with tunable compression ratio. This tuning is achieved by mixing incompressible and highly compressible data at appropriate and variable proportions. The shortcoming of this approach is that it only considers a single dimension —that of compressibility. It ignores the *time* required for actual compression and does not support *heterogeneity* of the data within files.

To exemplify this, we tested data created in *LinkBench* for its compression properties. Thus, using `zlib` we calculated the compression ratio of the original chunks [10], defined as $\frac{original\_size}{compressed\_size}$, to generate synthetic chunks of similar compressibility with LinkBench. The results, shown in Fig. 2, confirm that (i) compression ratios are fairly accurate but, unlike what happens with real data, insensitive to the compressor, and (ii) compression times are very inaccurate —affecting the system's performance.

Thus, we face a situation where most storage benchmarks generate unrealistic contents, whereas representative datasets cannot be shared with the community. This reflects a need for a *common substrate for generating realistic and reproducible benchmarking data*.

1.1 Our Contributions

We present Synthetic Data Generator (*SDGen*): an open and extensible framework for generating realistic storage benchmarking contents. *SDGen* is devised to produce data that *mimics* real-world data. In this paper we focus on mimicking compression related properties, but we view it as a wider framework that can be used for other properties as well. The framework consists of a pluggable architecture of *data generators and characterizations*. Basically, characterizations capture properties of datasets and are then used by data generators to create arbitrary amounts of similar synthetic data. A salient feature of *SDGen* is that researchers can *share dataset characterizations instead of actual contents* to generate realistic synthetic data in a reproducible manner.

The main features of our solution include:

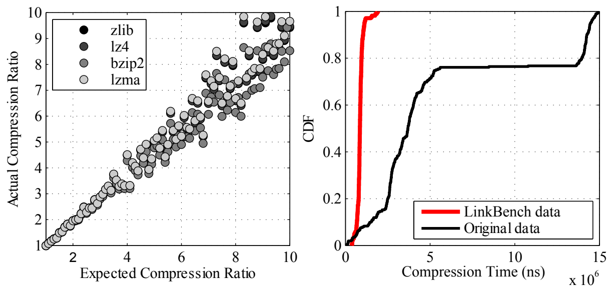


Figure 2: Compression ratios of LinkBench synthetic chunks for 4 compression engines (left). Compression times cumulative distribution function (CDF) of LinkBench data vs Canterbury corpus data (right).

Mimicking compression: Our first contribution is to identify the *properties of data* that are key to the performance of compressors, and therefore, to the performance of systems. Naturally, finding a universal solution to mimic data for *all* compressors is hard. The reason is that different compressors are guided by distinct heuristics in order to compress data. We therefore chose to focus on the most common category of compressors used in storage systems. Specifically, we target lossless compression that is either based on *byte level repetition finding* (Lempel-Ziv style algorithms [41, 42]) and/or on *entropy encoding* (e.g. Huffman encoding [20]).

As a second contribution, *SDGen* generates data that *mimics* both *compression ratios* and *compression times* of the original dataset for several compression engines. Moreover, our synthetic data exhibits similar *variability* of these parameters compared to the original datasets. Our tests exhibit that, on average, *SDGen* generates synthetic data for which compression ratio deviates less than 10% *and* its processing time deviates less than 15% from the original data. This was shown for disparate data types and with several different compression engines (`lz4`, `zlib-1`, `zlib-6`). For other compressors that vary in their core methods (`lzma`, `bzip2`) the results are less tight but also acceptable. We also verify the mimicking effect of working with our synthetic data over ZFS.

Compact and anonymized representation: Our third contribution is to design a *practical and private way of sharing benchmarking data*. *SDGen* users can reproduce a synthetic dataset by sharing a compact characterization file, being agnostic to the original dataset contents. This approach benefits from easy mobility coupled with the privacy of not sharing actual data. The characterization takes just 2.5KB per data chunk (for arbitrary chunk size, e.g. 64KB). We also explore the option to use random sampling to efficiently scan very large datasets, creating a constant size characterization while maintaining high accuracy in mimicking the entire dataset. In our tests, the overall characterization does not to exceed 8.5MB irrespective of the dataset size, which can be Terabytes.

Storage Domain	*Article/Benchmark*	*Data Generation Method*
File System	FileBench [5]	(**R**) Internally, FileBench gets random offsets of the internal memory to fill write buffers.
	Impressions [7]	(**C/D**) Binary files are zeroed whereas text files are filled with a static list of words sorted by popularity (English language). Impressions relies in third party applications to generate specific file types (mp3, jpeg).
Micro-benchmarks	IOzone [26]	(**R**) Random data that can be specified with a ratio of change per-operation to specify deduplication.
	VDBench [2]	(**M**) Compression ratio supported by mixing random and zero data and block repetition for deduplication.
	Bonnie++ [3]	(**R**) Write/update operations are filled with non-initialized char arrays.
	fio [4]	(**M**) Mix of random and zero data to fill IO operations with compression defined data.
Database/KV Store	OLTP [15]	(**R**) Payloads of queries are filled with random data.
	LinkBench [9]	(**M**) Static configuration of data compressibility that mixes new and existing data in each query.
	YCSB [13]	(**C**) The values of each table field are a string of ASCII characters of predefined length.
Cloud Storage	CloudCmp [22]	(**R**) Blobs for `put` operations are filled with random data.
	Drago et. al. [16]	(**R**) Benchmarking files are either random or with random words to emulate text.
	COSBench [39]	(**R**) API `put` operations are filled with random data.
	ssbench [1]	(**C**) Uploaded objects are filled with a single character (e.g. ‘A’).
Deduplication	Tarasov et. al. [33]	(**D**) Generation of deduplication workloads based on a Markov model. Initial contents are delegated to a first dataset image.
	DEDISBench [9]	(**D**) Initial contents are delegated to a first dataset image.

Generated contents are: (**R**)andom data, (**C**)onstant/zeros data, (**M**)ix of compressible and non-compressible data, (**D**)elegated to application/assumes initial dataset

Table 1: Data generation approaches for several widely adopted benchmarks in various storage domains.

Usability and integration: We plan to release *SDGen* to the community[1] as well as a set of public characterizations for some popular data types. Users can either use a public characterization, or create a new one in order to mimic their proprietary data. As a proof-of-concept, we also integrated *SDGen* as a data generation service into two well-known benchmarks suites: *Impressions* [7] (file system) and *LinkBench* [9] (social graph).

Paper organization: The rest of the paper is structured as follows. Section 2 discusses related work on synthetic data generation. Section 3 presents the *SDGen* architecture and in Section 4 we present our data characterization and generation methods for compression techniques. We evaluate *SDGen* in sections 5 and 6. In section 7 we describe the integration of *SDGen* with popular benchmarking tools. We draw some conclusions in Section 8.

2 Related Work

Benchmarking storage systems has long been an important research topic for the storage community. A vast amount of microbenchmarks and domain-specific benchmarks have been proposed in the last decade [5, 7, 9, 26, 28]. However, although these solutions provide flexible and realistic [8, 32] workload modeling, they do not consider the generated content as this is not their goal.

In Table 1, we summarize how many of these benchmarks generate data as stated in the official documentation or as we inferred by inspecting their source code. As mentioned in the introduction, some of these directly offer compression ratio tuning and some offer simple implementation also for deduplication ratio. We also refer the reader to [35] for an excellent overview of the state-of-the-art in storage benchmarking.

Tay [34] advocates for an application-specific benchmarking approach [29]. This work aims to augment an empirical dataset to an arbitrary size for database benchmarking. It provides a theoretical study of how to keep the internal database structure. For RDF databases Schmidt et. al. [28] suggested to include document and query generation modules based on a study of the DBLP system. Similarly to LinkBench [9], they emulate the behavior of users to guide the workload execution. Adir et al. presented an approach for benchmarking databases in which data is generated according to the customer's specifications in order to match his proprietary settings and data types [6]. They can optionally scan specified database columns to collect statistics on used words and use them in the benchmark.

Generating realistic contents for system benchmarking seems to be gaining momentum in the field of big data [36]. The authors of [36] propose BigDataBench, a complete benchmark for systems such as Hadoop and key-value stores. In particular, BigDataBench provides a data generation module that emulates predefined data types (e.g. text, graphs). In contrast, *SDGen* analyzes any given dataset to generate similar synthetic data. That is, a text file can be very compressible or not, depending on its contents. *SDGen* is able to capture this characteristic and generate synthetic data accordingly.

The closest work to the present paper we are aware of is that of Tarasov et. al. [33], which identified the relevance of generating realistic workloads for benchmarking deduplication systems. They propose a framework to capture and share the *updates* of datasets; traces representing *update operations* can be reproduced over other datasets to emulate deduplication. Clearly, *SDGen* shares the same spirit of [33]. However, in practice Tarasov et. al. delegate the actual dataset contents to an *initial image/snapshot*. *SDGen* fills this gap by providing synthetic initial contents similar to the original ones, which is a preliminary step to the deduplication benchmarking.

3 *SDGen*: Framework Architecture

SDGen is designed to capture characteristics of data that can affect the outcome of applying data reduction techniques on it. As we show next, *SDGen* works in two phases: A priming *scan* phase which build data characterizations to be used by a subsequent *generation* phase.

[1]Available at `https://github.com/iostackproject/SDGen`.

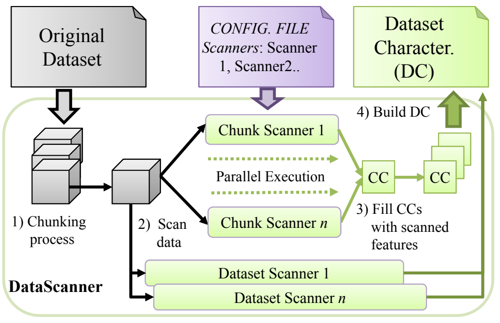


Figure 3: *SDGen* dataset scanning and characterization.

3.1 Scan Phase: Characterizations

To capture the characteristics of data, *SDGen* implements a two-level scan phase: *chunk level* and *dataset level*.

Many compression algorithms (e.g. `lz4`, `zlib`) partition the input data stream into chunks, and apply compression separately for every chunk [41]; such algorithms try to exploit redundancy which stems from locality of data (repetitions, common bytes) while minimizing the size of their internal data structures. Therefore, a central element in our design is the **chunk characterization** (*CC*). A CC is a user-defined module that contains the necessary information for every data chunk. *SDGen* scans a given dataset by splitting its contents into chunks (e.g., from 8KB to 128KB, configurable by the user) that are characterized individually (step 1 and 2, Fig. 3). We depict our CC design in Section 4.2.

In a higher level, *SDGen* builds **dataset characterizations** (DC), which provide a more holistic characterization. In the current version of *SDGen*, DCs store the deduplication ratio of the entire dataset as well as a list of all the previously generated CCs.

To support the above scans, *SDGen* applies two modules: `Chunk scanners` and `Dataset scanners`. These modules are loaded from the configuration in a manager class (`DataScanner`), which processes the dataset, and concurrently uses it as input for the scanners in order to build the characterization. The `DataScanner` life-cycle appears in Fig. 3.

The scan phase ends by persistently storing a DC (step 4, Fig. 3). *SDGen* also includes a way of transparently storing and loading DCs, enabling users to easily creating and sharing them.

3.2 Generation Phase

Once in possession of a DC, users may load it in *SDGen* to generate synthetic data similar to the original dataset.

The heart of the generation phase is the *generation algorithm*. This algorithm is designed by the user and receives as input a CC filled with the data characteristics captured by chunk scanners (see Section 4.3). Since CCs are read-only and independent of each other, the generation algorithm can utilize parallelism for faster data generation. A module called `DataProducer` orchestrates the content generation process. The `DataProducer` is also responsible for taking into account dataset-level characteristics during the generation process. Currently, this is mainly used for generating duplicated data. However, we concentrate on data compression, leaving the analysis of deduplicated data for future work.

The `DataProducer` module generates data using two API calls: `getSynData()` and `getSynData(size)`. The first call retrieves entire synthetic chunks with the same size as the original chunk. This is adequate for generating large amounts of content, such as file system images. The second call specifies the size of the synthetic data to be generated. This call is an optimization to avoid wasting synthetic data in benchmarks that require small amounts of data per operation (e.g. OLTP, databases). Technically, successive executions of this method will retrieve subparts of a synthetic chunk until it is exhausted and a new one is created.

3.3 Sampling at Chunk-level

SDGen generates a chunk characterization data structure for each data chunk. However, the time to scan a very large dataset can be prohibitively long and the size of the characterizations can grow excessively. To remedy this, we resort to sampling, i.e. scanning only a random fraction of a given dataset.

The crux is that random sampling is a good estimator for many properties of the data, and specifically for properties that can be expressed by averages and sums such as compression ratio or compression time. Harnik et. al. [18] show that using random sampling on chunks is a good estimator for compression ratio (within an additive percentage factor). The same also holds for estimating the fraction of data with specific compressibility or within specific compression time limits. Note that compression time of blocks is not bounded the way that compression ratio is. Compression time typically has higher variance, so typically its estimation is less tight than that of compression ratio. Still we argue, and corroborate through experimentation, that the sampling's accuracy is well within what is required for benchmarking.

The actual number of samples is a constant regardless of the size of the entire data set. In our tests we took $\sim 3,500$ chunks (this number meets accuracy guarantees provided in [18][2]), where each chunk in the data set is chosen with equal probability. This sampling can be done in a simple manner when dealing with large files or block devices, or by using the methodology of [17] for the case of file systems and other complex structures. For

[2]Sample size is set by *confidence* and *accuracy* parameters. *Accuracy* (we use the value 0.05) measures the additive distance that the estimation can vary from the actual compression ratio, while *confidence* (we use 10^{-6}) bounds the probability of falling outside this accuracy range (probability is taken over the randomness of the sampling).

each of the chosen chunks, a characterization is created and stored. In the data generation phase, data is created by taking characterizations in a round robin fashion. We test the accuracy of this sampling strategy in Section 6.6.

3.4 How to Extend *SDGen*

SDGen enables users to integrate novel data generation methods in the framework. To this end, one should follow three steps:

1. *Characterization*: Create a CC extending the `AbstractChunkCharacterization` class. This user-defined characterization should contain the required information for the data generation process.
2. *Scanners*: Provide the necessary scanners to fill the content of CCs and DCs during the scan process. Chunk-level scanners should extend from `AbstractChunkScanner` and implement the method `setInfo`, to set the appropriate CC fields.
3. *Generation*: Design a data generation algorithm according to the properties captured during the scan phase. This algorithm should be embedded in a module extending `AbstractDataGenerator`, to benefit from the parallel execution offered by `DataProducer`. Concretely, a user only needs to override the `fill(byte[])` method to fill with synthetic data the input array.

SDGen manages the life-cycle of the user-defined modules to scan/generate data, which are loaded from a simple configuration file. Finally, *SDGen* consists of 5,800 lines of Java code, including the framework architecture, our generation methodology (plus `Deflate` algorithm), the integration with LinkBench, and 200 lines of C++ code for integration with Impressions.

4 Compression-oriented Synthetic Data

Creating an efficient and accurate *mimicking method* is a non-trivial task (i.e. characterization, generation). In this section, we describe the research insights that guided the design of our method. We evaluate it in Section 6.

4.1 Generation Method Rationale

Mimicking data for compressors requires a basic understanding of how compressors work. In this work we target compressors that utilize repetition elimination to reduce data size. We also target compressors that use entropy coding (such as Huffman codes), typically on top of repetition elimination.

With this in mind, and based on empirical tests we identified two main characteristics that affect the performance and behavior of compression algorithms: *repetition length distribution* and *frequencies of bytes*.

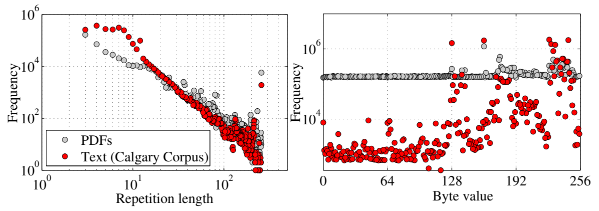


Figure 4: Repetition length distribution (left) and byte frequency (right) in PDFs and text data.

In repetition elimination a data byte is either represented by the byte itself (termed *literal*) or as part of a repetition. Each repetition is represented by its length and a back pointer (distance parameter). The repetition length is key since longer repetitions contribute to better compression ratio as well as to significantly better performance of compressors and decompressors. Note that the typical distribution of repetitions tends to follow a power-law [12], as observed in empirical tests (see Fig. 4 (left)). The majority of repetitions in such distributions are short ones (< 10 bytes) and consequently, compression algorithms exert effort in order to exploit these small repetitions, which in turn has an impact on performance. On the other hand we found that the effect of repetition distances on compression ratio and time is minor[3].

Entropy encodings utilize non-uniformity of byte level frequencies to encode data in a more compact representation at the bit level. In essence, the encoding associates bit level identifiers to bytes so that the most frequent bytes are represented by the shorter identifiers, saving storage space. This process is mimicked by capturing the distribution of bytes during the scan process. As we observe in Fig. 4 (right), the skew in the distribution of byte frequency changes significantly from text files to random-like data (PDFs). This has a strong impact on compressibility and may also impact the encoding process speed. These observations guided the design of our mimicking method for compression algorithms.

4.2 Data Characterization

To capture the aforementioned data characteristics, in our method every Chunk Characterization (CC) contains:

Byte frequency histogram. We build a histogram that relates the bytes that appear in a data chunk with their frequencies, encoding it as a `<byte, frequency>` map that we use to generate synthetic data that mimics this byte distribution. This information is key to emulate the entropy of the original data, among other aspects.

Repetition length histogram. Our aim is to mimic the distribution of lengths of repetitions as they would be found by a compressor. Note that different compressors will find different repetitions, depending on how much they invest in this task. Since there is no absolute answer here, we take a representative example of a compressor

[3]An entire data window typically fits in the L1 cache and thus a longer distance does not incur a performance penalty.

(default `zlib`'s `Deflate` algorithm) and work according to repetitions found by this compressor. To encode the repetitions as a histogram, we use a map whose keys represent the length of repetitions found in a chunk and the values are frequencies of repetitions of a given length.
Compression ratio. Every CC also includes the compression ratio of the original data chunk. In the generation phase, *SDGen* will try to create a synthetic chunk with similar compressibility.

Note that the CC design only reveals statistical properties of data, but not data itself. This provides a high degree of data privacy, as we discuss in Section 8.

Characterization space complexity: In our method, the space complexity of a CC data structure is *bounded* irrespective of the scan chunk size. Specifically, to represent a data chunk every CC contains 2 histograms whose data is stored in a map data structure. In these maps, keys can be encoded in *byte* data type (repetition length, bytes) whereas values are expressed as integers (32-bit integer). Therefore, in the worst case a single map requires 1,280 bytes (256 keys · 4 bytes/value). In addition, we add the compression ratio (64-bit double), as well as the inherited fields from `ChunkCharacterization` (*size*, 32-bit integer and *seed*, 64-bit long).

The data chunk size can be arbitrarily large and can be chosen according to the application at hand (see more in Section 5). As a rule, we suggest to make it at least as long as the size of a compressor's compression window (e.g., 32-128KB are typical granularities). Altogether, a CC consumes at most 3.93% the space of a 64KB chunk.

4.3 Synthetic Content Generation

In order to generate `dataSize` bytes of synthetic data (`synData`), we sequentially pick CCs from the list contained in the dataset characterization. The scanned features in every CC are the input values for our data generation algorithm, which is described in Algorithm 1.

Algorithm 1 generates synthetic data that mimics some key properties of the original data (see Section 4.2):
Byte frequency: Algorithm 1 generates both unique and repeated sequences with the function `randomData` that outputs random bytes based on the histogram of byte frequencies extracted from the original chunk (`byteFrq`).
Repetition length: We insert both random and repeated data in sequences of length `seqLen`, whose values are drawn by the repetition length histogram of the original chunk (`repLenFrq`→`seqLengths`). Normally, to generate repeated data we use a single sequence (`repSeq`) of length *MAX_LEN*. Thus, every time we need to insert a repetition, we select the first `seqLen` bytes of `repSeq`.
Compression ratio: For mimicking compressibility, Algorithm 1 interleaves repeated or unique sequences of bytes based on random trials (line 19) against the normalized compression ratio (`cr`) of the original chunk[4].

Algorithm 1: High-level data generation algorithm

```
   Data: dataSize, repLenFrq (Map), byteFrq (Map), cr
   Result: synData
1  $synData \leftarrow []$;
2  $uniqueBytes \leftarrow |byteFrq.keys()|$;
3  /*No need for renewal by default*/
4  $renewalRate \leftarrow \infty$;
5  $repCount \leftarrow 1$;
6  $i \leftarrow 0$;
7  /*Special treatment for extreme data types*/
8  if $uniqueBytes < MIN\_BYTES$ then
9      $renewalRate \leftarrow uniqueBytes$
10 /*Initialize repetition and set it as prefix*/
11 $repSeq \leftarrow randomData(byteFrq, MAX\_LEN)$;
12 /*Fill the repetitions distribution list*/
13 $seqLengths \leftarrow getDescOrderSeqLen(repLenFrq)$;
14 $previousWasRep \leftarrow False$;
15 while $i < dataSize$ do
16     if $seqLengths = []$ then
17         $seqLengths \leftarrow$
           $getDescOrderSeqLen(repLenFrq)$;
18     $seqLen \leftarrow seqLengths.popFirst()$;
19     if $randomTrial() < 1/cr$ then
20         $synData[i : i + seqLen] \leftarrow$
           $randomData(byteFrq, seqLen)$;
21         $previousWasRep \leftarrow False$;
22     else
23         /*Break to avoid repetition concatenation*/
24         if $previousWasRep$ then
25             $synData[i] \leftarrow randomData(byteFrq, 1)$;
26             $i \leftarrow i + 1$;
27         /*Add repeated data*/
28         $synData[i : i+seqLen] \leftarrow repSeq[0 : seqLen]$;
29         /*Renew repetition if necessary*/
30         if $(repCount$ mod $renewalRate) = 0$ then
31             $repSeq \leftarrow$
               $randomData(byteFrq, MAX\_LEN)$;
32         $repCount \leftarrow repCount + 1$;
33         $previousWasRep \leftarrow True$;
34     $i \leftarrow i + seqLen$;
```

Algorithm 1 includes several implementation nuances resulting from our empirical insights. First, this algorithm generates batches of repeated/unique byte sequences that are appended to the synthetic chunk in decreasing order by length (`getDescOrderSeqLen` in lines 13, 17). This choice allows algorithms with light repetition search (e.g. `lz4`) to find the correct synthetic repetitions. We empirically found that compressors with deeper repetitions search are insensitive to such ordering.

[4]Since we employ the `zlib` repetition finding algorithm, we also make use of the `zlib` compression ratio in our characterization. `zlib` sets the maximum repetition length at 258 (*MAX_LEN*, Algorithm 1).

Second, we avoid the concatenation of several repetitions by adding a random separator byte after every repetition (line 25). The reason is that two consecutive repetitions of the same length may occur more than once. Given that we normally use a single sequence as a source of repetitions (`repSeq`), these concatenations would be interpreted as a single longer repetition.

Note that our method is susceptible to mutual effects caused by the interplay between repetitions and byte distribution. Repetitions can slightly skew the byte distribution while a short alphabet can also affect the repetition counts (typically making them longer). In general, our evaluation showed that such these interplays have a minor effect on other metrics. An exception is with data formed with a very short alphabet (e.g. DNA sequencing)[5] which require special treatment. Algorithms like `zlib` exhibit degraded results on such data, probably because the internal Huffman tree should be constantly updated for very short sequences. To overcome this problem, we refresh the repeated sequence often during the generation process (`renewalRate`), to degrade the performance of compressors as in the original data (line 9).

Albeit simple, in Section 6 we show that our method provides an attractive trade-off between characterization complexity and accuracy, for disparate datasets. We also show that the accuracy of our synthetic data, in terms of compression ratios and times, is because it mimics the key properties defined in our characterizations (e.g., repetition lengths, byte distribution).

5 Experimental Setup

For clarity, our evaluation is divided into two parts: i) evaluation of the accuracy of the synthetic data generated by *SDGen* (Section 6), and ii) analysis of the benefits of *SDGen* integrated with real benchmarks (Section 7).

Methodology. To quantify the accuracy of the synthetic data that *SDGen* generates, we proceeded as follows. First, we scanned an original dataset in fixed size chunks (32KB) to build a full characterization file. For practicality, the scan process was done over a single `.tar` file containing all the files of a dataset. Subsequently, we generated a synthetic file as large as the original one.

Then, we analyzed the behavior of compression engines (compression ratios, times) on a *dataset and per-chunk basis*. For inspecting chunks, we instantiated a fresh compressor object to digest every data chunk. Compressors were executed sequentially, to avoid artifacts and interferences in compression times. This fine-grained perspective enabled us to capture the heterogeneity of datasets and the reaction of compression engines. Dataset compression times are averages of 30 executions.

We compared *SDGen* with LinkBench data generation, which is a representative case of solutions to generate data with predefined compressibility by simply mixing compressible/incompressible sequences (see Table 1). To this end, we used `zlib` to obtain the compression ratio of the original chunks. We then generated similarly compressible data chunks with LinkBench, using its default data generation mechanism. Note that this goes far beyond the standard implementation, which targets a preconfigured mean data compressibility.

[5]We empirically found alphabet size 8 to be a good threshold value.

Setting. The evaluation was performed using a server running a Debian 7.4 operating system, equipped with an i5-3470 processor (4 cores), 8GB DDR-3 memory and a HDD of $7,200$ rpm and 1TB of storage capacity. Since SSDs are becoming increasingly popular for databases, in the integration tests of LinkBench we used a Samsung 840 SSD with 250GB of storage capacity.

We ran our sampling experiments on an Ubuntu 14.04 server equipped with an Intel Xeon x5570 processor (4 cores) with 8GB RAM. We read the files from a local disk, compressed them and stored them to an 8-disk raid-10 array (10K RPM SAS drives), via a 4Gbit FC port.

5.1 Compression Similarity Metrics

We evaluate the accuracy of our synthetic generation method by targeting several metrics. First and foremost, we aim to hit the two main parameters that are relevant to the performance of a system:

- **Compression ratio**: This metric refers to the ratio between the size of the original data to the size of the compressed data. Thus, given a compression algorithm, we want that a chunk of synthetic data will be as compressible as the original data chunk.
- **Compression time**: This metric captures the computation time taken by a compression algorithm to compress a single data chunk.

We also analyze two properties that helped us to devise our generation algorithm. They can serve as good indicators to the potential success of *SDGen* in mimicking data for other compressors that are not tested here:

- **Repetition length**: We compare the length of repetitions in both the original and synthetic data.
- **Entropy**: It is often associated with the compressibility of data [18] and quantifies how uniformly distributed the bytes are within a data chunk. From a sample $X = \{x_0, ..., x_n\}$ of byte values with a probability mass function $P(X)$, we calculate its entropy $H(X) = -\sum P(x_i) log_2 P(x_i)$ [30]. In a byte basis, an entropy value of 8 (highest) means that the data is completely random and therefore not compressible. A value of 0 means the opposite.

In addition, we performed several experiments over ZFS, a well-known file-system with built-in compression to measure transfer throughput. These experiments helped us to compare and evaluate the behavior of a real system when using synthetic/original data [23].

		Compression Ratio				*Compression Time (secs.)*			
Dataset	*Source*	`zlib`	`lz4`	`bzip2`	`lzma`	`zlib`	`lz4`	`bzip2`	`lzma`
Calgary C.	OR	3.30	1.80	4.03	4.25	1.128	0.082	1.692	10.251
	SD	3.15(−4.5%)	1.82(1.26%)	3.10(−23.29%)	3.58(−15.76%)	1.096(−2.8%)	0.072(−12.19%)	1.554(−8.15%)	9.064(−11.67%)
	LB	3.70(12.3%)	2.79(55.39%)	3.8(5.48%)	3.71(−12.72%)	0.329(−70.8%)	0.043(−47.57%)	3.113(83.98%)	4.350(−57.56%
PDFs	OR	1.16	1.15	1.15	1.19	1.431	0.109	6.554	15.758
	SD	1.13(−2.59%)	1.10(−4.34%)	1.10(−4.34%)	1.12(−5.88%)	1.551(8.38%)	0.117(7.33%)	6.514(−0.61%)	15.783(0.15%)
	LB	1.46(25.85%)	1.12(−2.61%)	1.47(27.82%)	1.46(22.69%)	1.658(15.86%)	0.106(−2.75%)	5.086(−22.39%)	18.526(17.56%)
Media	OR	1.01	1.00	1.01	1.01	10.809	1.971	40.18	113.38
	SD	1.00(−0.99%)	1.00(0%)	1.00(−0.99%)	1.00(−0.99%)	11.083(2.53%)	1.960(0.56%)	43.02(7.07%)	110.895(2.19%)
	LB	1.30(28.71%)	1.00(0%)	1.30(28.71%)	1.29(27.72%)	11.601(7.32%)	1.941(1.52%)	30.12(−25.04%)	129.87(14.54%)
Silesia C.	OR	3.11	2.08	3.88	4.31	8.214	0.902	21.022	102.82
	SD	2.82(−9.22%)	2.03(−2.40%)	2.73(−29.63%)	3.32(−22.97%)	8.826(7.45%)	0.833(−7.65%)	19.200(−8.67%)	82.33(−19.92%)
	LB	3.57(14.79%)	2.69(29.32%)	3.65(−5.93%)	3.56(−17.40%)	4.002(−51.27%)	0.476(−47.23%)	35.202(67.45%)	52.07(−49.35%)
Sensor	OR	4.86	2.88	5.93	8.33	55.014	17.795	121.03	1238.3
	SD	4.52(−6.99%)	2.70(−6.25%)	4.80(−19.05%)	5.67(−31.93%)	53.799(−2.21%)	16.643(−6.47%)	145.65(20.30%)	909.2(−26.57%)
	LB	5.32(9.46%)	3.99(38.54%)	5.54(−6.57%)	5.34(−35.89%)	26.659(−51.54)	12.486(−29.83)	304.96(151.9%)	381.6(−69.18%)

OR=Original Dataset, **SD**=*SDGen* Synthetic Data, **LB**=LinkBench Synthetic Data. Relative errors compared to **OR** appear in parentheses.

Table 2: Dataset-level compression ratios and times of *SDGen* and LinkBench data (non-sampling datasets).

5.2 Datasets and Compression Engines

Next, we briefly describe the datasets used to assess the accuracy of our data generation method. Note that we stress the importance for a method to be accurate in the presence of diverse and heterogeneous datasets.

- **Calgary/Canterbury corpus (Text)**: Collection of text and binary data files, commonly used for comparing data compression algorithms (18.5MB).
- **PDFs**: Proceedings of the last 5 editions of the Usenix FAST conference (48.7MB).
- **Silesia corpus**: Standard set of files that covers the typical data types used nowadays [14] (211.9MB).
- **Media**: Media files collected from the home directories of 4 IBM engineers including photos (`.jpg`), music (`.mp3`) and video (`.avi`) (300.3MB).
- **Sensors dataset**: GPS trajectory dataset collected in Microsoft Research Geolife project by 182 users during three years [40](1.7GB).
- **Mix** (*sampling test only*): A private collection of mix of files of various data types e.g. html, xml, txt, database files and VM images (14GB).
- **Enwiki9** (*sampling test only*): Common measuring stick for compression methods consisting of the first 10^9 bytes of the English Wikipedia [37].

We measured the accuracy of our synthetic data mimicking these datasets by analyzing the behavior of 4 compression engines: `lz4`, `zlib` (level 1, 6), `bzip2` and `lzma`. We used the compressors' default implementation available on Unix distributions and the analogous Java libraries included in *SDGen*. It is worth mentioning that these algorithms belong to *different families and adopt disparate heuristics* to find redundancies within a data stream. This can give a sense about the universality of our generation method. Specifically, `lz4` targets speed over compression ratio and has repetition elimination only. `zlib` is medium speed, adopting Huffman encoding in addition. `bzip2` and `lzma` target compression ratio over speed deploying various advanced, yet time consuming methods such as the Burrows Wheeler transform [24] (`bzip2`) or a large dictionary based variant of LZ77 (`lzma`) .

6 Evaluation

Next, we describe the results comparing original and synthetic (*SDGen*, LinkBench) data using the aforementioned metrics. We compare the results of two compression engines that we *specifically target* (`zlib`, `lz4`) with other two of distinct families (`bzip2`, `lzma`).

6.1 Compression ratio

In Table 2 and Fig. 5, we compare the obtained compression ratios of the original and synthetic datasets for all compression engines at both dataset and chunk levels.

Table 2 shows that *SDGen* closely mimics the compressibility of real datasets for the targeted engines (`zlib`, `lz4`). For `zlib` and `lz4`, *SDGen* does not deviate more than 10% in compression ratio compared to the real data. This demonstrates that our synthetic data is *sensitive to the algorithm*; that is, our synthetic data exhibits the same behavior than the real data, depending on the compression engine used. This confirms that analyzing the *structure of data* is an appropriate approach to generate realistic synthetic data.

At the dataset level, we observe that *SDGen* is less accurate for the non-targeted compressors (`bzip2`, `lzma`). Surprisingly, this contrasts with Fig. 5 that shows how *SDGen* closely reproduces the compression ratio distributions of real dataset chunks. The reason for this behavior is related with the *chunk size*.

That is, `zlib` and `lz4` digest data in small chunks (e.g. 32KB) to reduce the size of their internal data structures. Conversely, `bzip2` and `lzma` digest data in window sizes that can reach 900KB and 1GB, respectively [23]. We do not focus on scanning data features in this broader scope. This encourages us to research new scanners at larger granularities to cope with other compressors.

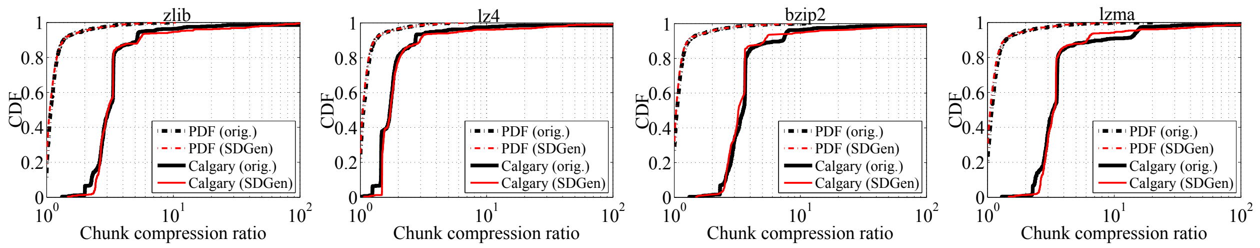


Figure 5: Original vs *SDGen* per-chunk compression ratio distributions for various datasets and compressors.

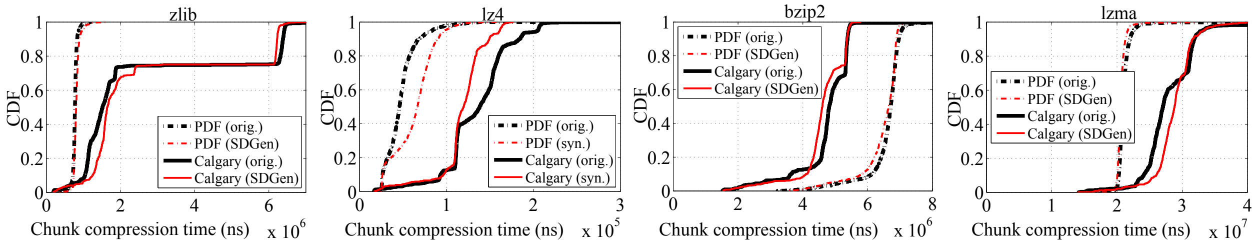


Figure 6: Original vs *SDGen* per-chunk compression time distributions for various datasets and compressors.

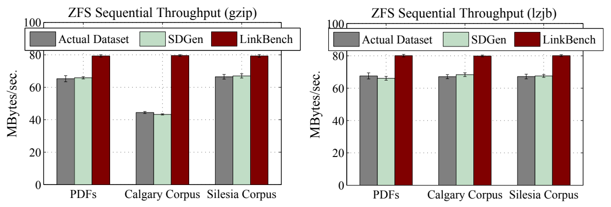


Figure 7: Sequential write throughput of ZFS depending on data type. Clearly, ZFS behaves similarly when processing *SDGen* synthetic data and the original one.

In general, the one-dimensional data generation approach of LinkBench deviates importantly from the compressibility of the actual data, even though LinkBench datasets were generated chunk-by-chunk to capture the heterogeneity (Table 2). The reason is that the proportion of compressible/incompressible data is determined by one algorithm in the generation phase (`zlib`). Such a synthetic data becomes inaccurate when compressed by other engines, compared to the original dataset.

6.2 Compression time

Table 2 shows that *SDGen* achieves high accuracy mimicking compression times for `zlib` and `lz4`. At the dataset level, our synthetic data does not deviate by more than 13% in compression times compared to the original data. At chunk level, on average the 70% of *SDGen* chunks deviate less than 20% in terms of compression time (Fig. 6) w.r.t. the real dataset —it is harder to be accurate for `lz4` since it is the fastest compressor. Table 2 also illustrates that for the non-targeted compressors (`bzip2`, `lzma`) our synthetic data does not deviate by more than 26%, which we consider acceptable.

Note that in the Calgary corpus there is a particular file that accounts for the 23% of samples, which exhibit a much higher compression time for `zlib` (Fig. 6, left). This file is the DNA sequencing of the E. coli bacteria. The particularity of this file is that it is formed by very few distinct bytes (4 in most chunks, since DNA sequences are composed by 4 nucleotides) and very short repetitions. This makes compression algorithms that use Huffman codes to perform worse, since the Huffman tree should be constantly updated for only very short sequences. Our generation algorithm detects these situations and reacts by increasing the repeated sequence renewal rate. Thus, the performance of Huffman codes becomes also worse, similar to the original data.

Unsurprisingly, Table 2 shows that LinkBench datasets deviate importantly from the compression times of real datasets. Such a deviation —in many cases higher than 50%— may induce important impact on the performance of a storage system with built-in compression.

6.3 Performance of ZFS

We want to stress the importance of using an appropriate data generation method in a real system. Thus, we augmented 3 datasets (PDFs, Calgary and Silesia) by replicating them to be 1GB in size. Then, we copied 50 times each from memory into a ZFS partition with compression enabled (`lzjb`, `gzip`) capturing the sequential write throughput. We repeated the experiment with two synthetic datasets: i) a dataset generated with LinkBench creating chunks of the same compressibility than the original one, and ii) a dataset generated with our method.

In Fig. 7, our synthetic data makes the system to exhibit virtually the same write throughput as the original dataset. That is, the difference in throughput of ZFS between the original dataset and our data is at most 1.9% in average for `lzjb` and `gzip`. However, considering datasets generated with LinkBench, ZFS exhibits a variation in write throughput between +12.5% and +19% in most cases w.r.t. the original dataset. Interest-

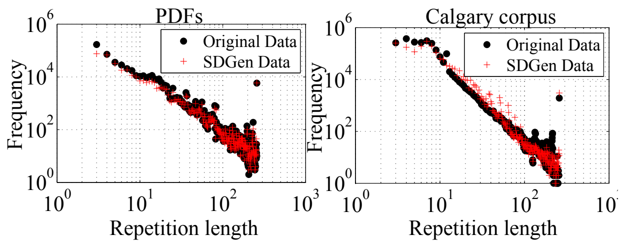


Figure 8: Repetition length distributions of *SDGen* and original datasets.

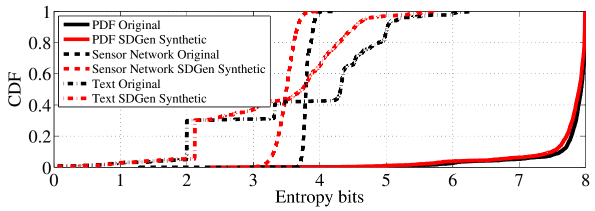


Figure 9: Entropy CDF of *SDGen* and original data.

ingly, irrespective of the dataset, ZFS achieves a similar throughput writing LinkBench data. The reason is that LinkBench generates data that is easy to compress for most algorithms, for a wide range of compression ratios.

`gzip` (as `zlib` in Fig. 6) performs worse digesting the Calgary corpus. Our method handles this behavior, whereas the LinkBench data makes ZFS write throughput to deviate by +44% compared to the original dataset.

Therefore, compared with current approaches, our method provides a much more realistic substrate to benchmark storage systems with compression built-in.

6.4 Similarity of Mimicked Characteristics

To emulate compression ratios and times, our mimicking method captures the *repetition length distribution* and the *frequencies of bytes* of the original data (Section 4). Now, we inspect how close *SDGen* mimics these properties.

Repetition length. Fig. 8 shows the distribution of repetition lengths for both original and synthetic datasets (PDFs, text). We observe that the repetitions in both cases are similar in terms of distribution shape and absolute frequency numbers. This characteristic plays a key role on the accuracy of the synthetic chunks compression ratios and times, suggesting that mimicking it is an effective way of generating realistic data for compression.

Entropy. In Fig. 9 we depict the entropy distribution of original and synthetic chunks for two datasets. As we can infer, the entropy distribution of the original dataset is roughly followed by the synthetic one. The reason for this is that we capture the byte histogram distribution in original chunks to generate bytes according to it. This property is also interesting because our synthetic data would be useful for techniques that estimate the compressibility of data based on entropy [18].

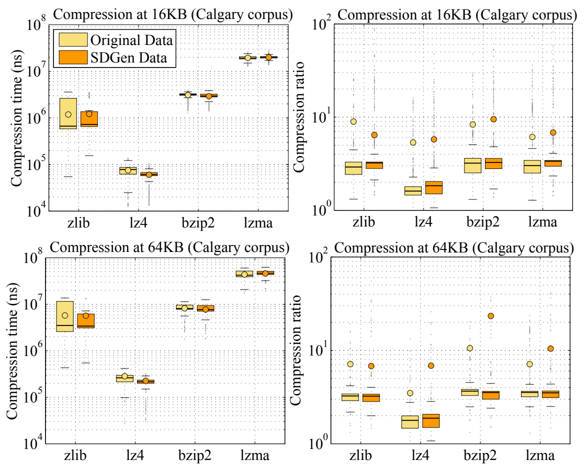


Figure 10: Compression times and ratios of 16KB and 64KB synthetic chunks mimicking Calgary Corpus data. Original data was scanned at 32KB chunk granularity (boxplot circles represent mean values).

6.5 Chunk Size Sensitivity

We obtained very similar results when varying the chunk size from 8KB to 128KB, which are the typical size limits for compression (16KB for MySQL, 8KB to 128KB for ZFS). However, as the scan chunk size gets smaller, the characterization file grows linearly. Normally, compression algorithms tend to avoid small window sizes since data compressibility decays. Thus, we recommend to scan data in chunks of 16KB to 64KB.

In *SDGen* we scan a dataset and generate data with a configurable chunk size. A question that arises here is: *how does the synthetic data behave when it is compressed to different granularities than the one used in the scan/generation process?* To answer this question, we compressed the Calgary corpus and the synthetic dataset at 16KB and 64KB granularities (Fig. 10). Note that the synthetic dataset has been scanned/generated at 32KB granularity. We selected the Calgary corpus since it is very heterogeneous (compression times, ratios), potentiating differences depending on the scan granularity.

Interestingly, in Fig. 10 we observe that our method is not sensitive to the scan granularity. That is, for both scan chunk sizes, the compression times and ratios of compression algorithms follow a similar trend. Although the distribution tails are harder to model, we observe that in most cases the boxplots for both datasets present a similar shape. Therefore, we conclude that we can safely mimic a dataset using granularities that are different than the one used during the original data scan phase, while maintaining the original content behavior.

6.6 Sampling: Scaling Characterizations

Previously, we performed full scans on the original data (32KB chunks). Full dataset scans let us reproduce the compressibility of data, and even the locality of compres-

Dataset	Compressor	Compression Time (sec.)				Decompression Time (sec.)				Compression Ratio			
		Orig.	*SDGen*	Zero	Rand.	Orig.	*SDGen*	Zero	Rand.	Orig.	*SDGen*	Zero	Rand.
Mix	`gzip-6`	443.47	451.98	136.48	589.82	97.30	95.74	72.52	95.64	1.86	1.82	1030.5	1.00
	`gzip-1`	378.28	380.45	136.34	574.46	88.30	87.21	48.64	96.32	1.84	1.82	229.3	1.00
	`lz4`	161.82	143.21	137.15	139.93	42.11	43.50	5.98	66.63	1.79	1.78	254.7	1.00
Enwiki9	`gzip-6`	50.31	57.16	9.41	39.32	8.20	9.05	4.84	6.32	3.09	2.83	1030.5	1.00
	`gzip-1`	22.08	22.92	9.45	37.93	9.18	9.27	6.05	3.23	2.64	2.54	229.3	1.00
	`lz4`	10.00	9.60	9.40	9.39	3.20	2.93	0.41	3.92	1.97	1.91	254.7	1.00

Orig=Original Dataset, **SDGen**=*SDGen* Dataset, **Zero**=Zero Dataset, **Rand**=Random Dataset.

Table 3: Dataset-level compression ratios and times of *SDGen* data using sampling in the scan process.

sion with high precision. In fact, this can be achieved with moderate characterization space requirements. For instance, compared to the original datasets, the size of characterization files are 4.08% (Silesia corpus), 2.14% (Calgary corpus), and 6.38% (PDFs) —the theoretical maximum is 7.86% for a 32KB chunk. However, considering large datasets, the size of characterizations and the memory requirements for the scan may be too high.

Next, we want to inspect the accuracy of our synthetic data when we only use a subset of data in the scan process. To this end, we make use of real datasets that are large enough to justify the use of sampling. As described in Section 3.3, we use characterizations formed by $3,500$ samples. We also tested a larger chunk size (128KB).

Table 3 shows the compression/decompression times, as well as the compression ratios for the real and synthetic datasets. "Random" and "Zeros" are datasets as large as the real one, whose content is self-explanatory.

In general, we see in Table 3 that our sampling approach is an effective way of mimicking large datasets. That is, *SDGen* datasets do not deviate more than 13.6% and 10% in compression and decompression times, respectively. *SDGen* compression ratios are also accurate.

Interestingly, we find that decompression times are similar in both original and *SDGen* datasets; this suggest that if the synthetic data mimics correctly compression times, decompression times become also mimicked.

The most relevant point in this experiment is the size of the characterizations needed to achieve these results. That is, the Mix dataset (14GB) characterization file produced by *SDGen* was only 7.3MB in size (0.052%). We conclude that *SDGen* provides a novel and attractive way of sharing large datasets with very low data exchange, high mimicking accuracy and preserving data anonymity.

6.7 Data Generation Throughput

We evaluate the throughput of *SDGen* generating synthetic data with our method. First, *SDGen* utilizes the available cores to increase the generation throughput (Fig. 11). That is, making use of 4 cores instead of 1, the throughput of *SDGen* is x3.56 and x3.45 times higher in the case of text and media data, respectively.

Second, we noticed that the generation throughput varies depending on the data type being generated. This effect is caused by the behavior of our data generation method. That is, Algorithm 1 is faster generating highly compressible data since it reuses repetitions more frequently to generate synthetic chunks. In the case of random-like data, Algorithm 1 performs more random decisions to generate new bytes, which is slower than copying an existing sequence. Note that several *SDGen* servers can run in parallel to increase throughput.

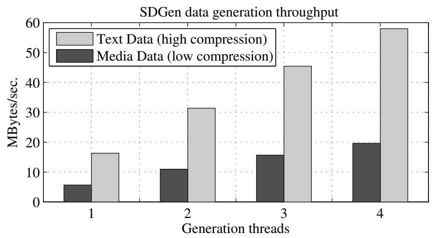


Figure 11: Throughput of our data generation algorithm integrated in *SDGen* depending on the data type.

7 Integration with Benchmarks

7.1 LinkBench

LinkBench [9] is a graph benchmarking tool developed by Facebook. LinkBench provides performance predictions on databases used for persistent storage of Facebook's production data. We integrated *SDGen* with LinkBench to explore the benefits of our synthetic data.

Integration. Internally, LinkBench permits to adjust the compressibility of data used in every experiment run. To decouple the actual data generation from the benchmark execution, LinkBench provides an interface called `DataGeneration`. We followed this contract by creating an adapter class which transforms LinkBench calls into API calls offered by our data generation system. This clean design permits a user to choose the way synthetic data is generated from the configuration file.

Setting. We executed LinkBench on top of MySQL 5.5 and ZFS with compression enabled (`lzjb` and `gzip`). We evaluated the differences in performance that are measured by LinkBench when query payloads are filled with realistic and non-realistic synthetic data given a target dataset (text, Calgary corpus). We filled query payloads with random offsets of datasets loaded in memory prior to the benchmark execution, to avoid the potential bias caused by the generation overhead.

We executed a write-dominated workload to observe the effects of content on insert latencies. We evaluated the performance of inserts due to their higher cost compared to reads, since they cannot be cached. We executed

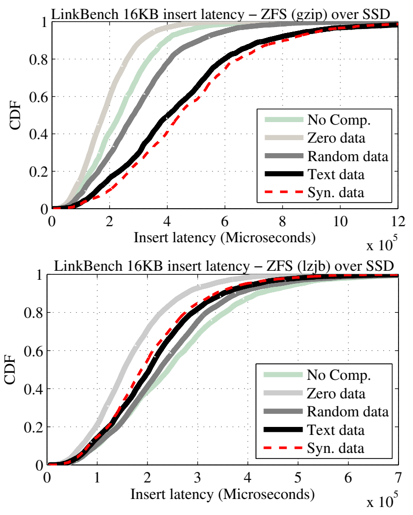


Figure 12: Insert latency of LinkBench for 16KB payloads on top of ZFS with `gzip` and `lzjb` compression.

LinkBench with 100 parallel threads for 30 minutes, resulting in various millions of inserts (18GB database). We used a Samsung 840 SSD as storage layer (250GB).

Results. In Fig. 12, we illustrate the insert latency that LinkBench experiences depending on the payload content. First, we want to emphasize the benefits of *SDGen* compared to use non-realistic data. Observably, the insert latency distributions are very similar for both the Calgary corpus and the corresponding *SDGen* synthetic dataset. This probes that *SDGen* produces *representative and reproducible data*, since the Calgary corpus characterization is ready to be shared and serve others.

Furthermore, we observe that employing naive contents may lead to disparate performance results. For instance, the median insert time of LinkBench is -55% filling payloads with zeros than using the corpus (`gzip`). `lzjb` also presents such performance variations, but they are less significant since the algorithm is much faster.

7.2 Impressions

File systems are an important field to apply data reduction techniques. Thus, we integrated our data generation system in the Impressions file system benchmark [7].

Integration. The integration has been done as follows. We set up a named pipe during the initial phase of the execution of Impressions to connect with *SDGen*. From that point onwards, Impressions delegates the generation of file contents to *SDGen* by writing in the pipe the size and the canonical path of the files to be created.

Additionally, we added a special type of dataset characterization (DC) in *SDGen* called *file system characterization* (Fig. 13). A file system characterization internally contains a set of regular DCs, each one associated to what we call a *file category*. File categories represent a group of file types —based on their extension— that usually contain similar contents. For instance, we can define a file category called "random data" containing compressed/encrypted files, such as `.zip`, `.mp3` and `.jpg`. We found this approach very convenient to treat the vast amount of existing file extensions, also produced by Impressions. We provide an initial set of file categories grouping the most popular file extensions[6].

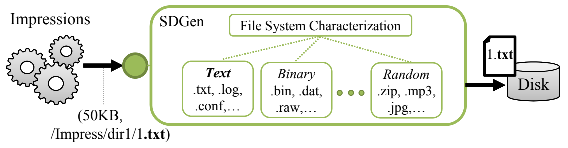


Figure 13: Integration of *SDGen* with Impressions.

To illustrate this, we recommend to see Fig. 13. In this figure, Impressions notifies *SDGen* that a new file called `/Impress/dir1/1.txt` of size 50KB should be created. *SDGen* looks up for the file extension `.txt`, which belongs to the "Text" file category. Subsequently, *SDGen* generates the file content resorting to the DC of this file category, which should be built by scanning representative files with the corresponding extensions.

Open Trial. We release a ready-to-use trial of *SDGen* integrated with Impressions. In the *SDGen* webpage, we release various DCs to be loaded into the file system characterization, which internally associates them to the appropriate file category. These DCs come from scanning file systems of our company engineers (text, images, etc.). We also provide the modified code of Impressions, as well as the execution instructions.

8 Discussion and Conclusions

A word on dataset privacy leakage: One of the main concerns in data sharing, or lack thereof, is privacy limitations on proprietary data. Our method relieves many of the privacy concerns and allows free sharing of data since *no actual data is shared*. More precisely, the data being created is a random combination of bytes, albeit with specific probabilities on byte occurrences and repetitions. It should be noted, however, that the characterizations are not entirely free of information. That is, the frequencies of bytes are revealed and these can actually tell us information about the data at hand. For instance, using such information one can distinguish the underlying *data type* (e.g. text, image).

Inherently, the mimicking approach is susceptible to this sort of "higher order information leakage" since the properties we are characterizing are *per se* an indication about the data at hand. However, we believe that this novel way of sharing provides an *attractive trade-off* between dataset privacy and benchmarking accuracy.

Beyond compression: In this paper we focused on compression related properties, but view *mimicking* as

[6]`http://en.wikipedia.org/wiki/List_of_file_formats`.

a general approach of data generation for benchmarking content-sensitive systems. One obvious extension is deduplication which is extremely data sensitive. Deduplication is more challenging for a number of reasons: (i) It is a *global dataset property*, rather than being dictated by local behavior of data. For this reason, fast scanning using sampling is much harder to achieve when deduplication is involved [19]; and (ii) deduplication ratios achieved in storage systems are typically affected by the *order and timing* in which data was written to the system (and not only by the content). Such a time based dependency is very hard to mimic. Our future research includes extending *SDGen* to also mimic data de-duplicability.

Conclusions: Most workload generators for benchmarks do not focus on the contents used in their execution, and they typically generate unrealistic data (zeros, random data). Storage systems with *built-in compression* behave differently on such naively-synthesized data than they do on real-world data. Current solutions create data with variable compression ratio, but they ignore other properties such as *compression time and heterogeneity*, which are critical to the performance of these systems.

We have therefore extended the basic methodology that underlies workload generators to the data itself. Current workload generators try to *mimic* real-world situations in terms of files, offsets, read/write balance and so on; we have designed and implemented an orthogonal component, called *SDGen*, to generate data that mimics the compressibility and compression times of real data.

For mimicking real-world data *SDGen* produces characterizations that are compact, sharable and essentially completely anonymized. We plan to release both *SDGen* and the characterizations that we have produced with it. We hope that others will use these tools and that others will share additional characterizations of data with the systems research community.

Acknowledgements

We first thank our shepherd Xiaosong Ma and anonymous reviewers. This work has been partly funded by the EU projects *CloudSpaces* (FP7-317555) and *IOStack* (H2020-644182) and Spanish research projects *DELFIN* (TIN-2010-20140-C03-03) and *Cloud Services and Community Clouds* (TIN2013-47245-C2-2-R) funded by the Ministry of Science and Innovation.

References

[1] Ssbench: Benchmarking tool for swift clusters. https://github.com/swiftstack/ssbench.

[2] Vdbench users guide. www.oracle.com/technetwork/server-storage/vdbench-1901683.pdf.

[3] Bonnie++. http://www.coker.com.au/bonnie++/, 2001.

[4] Fio. http://freecode.com/projects/fio, 2005.

[5] Filebench. http://sourceforge.net/projects/filebench/, 2008.

[6] ADIR, A., LEVY, R., AND SALMAN, T. Dynamic test data generation for data intensive applications. In *7th International Haifa Verification Conference, HVC 2011* (2011), pp. 219–233.

[7] AGRAWAL, N., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Generating realistic impressions for file-system benchmarking. In *USENIX FAST '09* (2009), pp. 125–138.

[8] ANDERSON, E., KALLAHALLA, M., UYSAL, M., AND SWAMINATHAN, R. Buttress: A toolkit for flexible and high fidelity i/o benchmarking. In *USENIX FAST'04* (2004), pp. 4–4.

[9] ARMSTRONG, T. G., PONNEKANTI, V., BORTHAKUR, D., AND CALLAGHAN, M. LinkBench: a database benchmark based on the facebook social graph. In *ACM SIGMOD'13* (2013), pp. 1185–1196.

[10] BELL, T. Canterbury corpus. http://corpus.canterbury.ac.nz, 1997.

[11] BURROWS, M., JERIAN, C., LAMPSON, B., AND MANN, T. On-line data compression in a log-structured file system. In *ACM ASPLOS'92* (1992), pp. 2–9.

[12] CLAUSET, A., SHALIZI, C. R., AND NEWMAN, M. E. Power-law distributions in empirical data. *SIAM review 51*, 4 (2009), 661–703.

[13] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *ACM SoCC'10* (2010), pp. 143–154.

[14] DEOROWICZ, S. Silesia corpus. http://www.data-compression.info/Corpora/SilesiaCorpus/, 2003.

[15] DIFALLAH, D. E., PAVLO, A., CURINO, C., AND CUDRE-MAUROUX, P. OLTP-Bench: An extensible testbed for benchmarking relational databases. *VLDB Endowment 7*, 4 (2013).

[16] DRAGO, I., BOCCHI, E., MELLIA, M., SLATMAN, H., AND PRAS, A. Benchmarking personal cloud storage. In *ACM SIGCOMM IMC'13* (2013), pp. 205–212.

[17] GOLDBERG, G., HARNIK, D., AND SOTNIKOV, D. The case for sampling on very large file systems. In *IEEE MSST'14* (2014), pp. 1–11.

[18] HARNIK, D., KAT, R., SOTNIKOV, D., TRAEGER, A., AND MARGALIT, O. To zip or not to zip: Effective resource usage for real-time compression. In *USENIX FAST'13* (2013).

[19] HARNIK, D., MARGALIT, O., NAOR, D., SOTNIKOV, D., AND VERNIK, G. Estimation of deduplication ratios in large data sets. In *IEEE MSST'12* (2012), pp. 1–11.

[20] HUFFMAN, D. A. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the Institute of Radio Engineers 40*, 9 (September 1952), 1098–1101.

[21] J. TATE, B. TUV-EL, J. Q. E. T., AND WHYTE, B. Real-time compression in SAN volume controller and Storwize V7000. Tech. rep., REDP-4859-00. IBM, 2012.

[22] LI, A., YANG, X., KANDULA, S., AND ZHANG, M. CloudCmp: comparing public cloud providers. In *ACM SIGCOMM IMC'10* (2010), pp. 1–14.

[23] LIN, X., LU, G., DOUGLIS, F., SHILANE, P., AND WALLACE, G. Migratory compression: coarse-grained data reordering to improve compressibility. In *USENIX FAST'14* (2014), pp. 257–271.

[24] MICHAEL BURROWS AND DAVID WHEELER. A block-sorting lossless data compression algorithm. Tech report.

[25] NIMBLE STORAGE. Nimble storage: Engineered for efficiency. Tech. rep., WP-EFE-0812, Nimble Storage, 2012.

[26] NORCOTT, W. D., AND CAPPS, D. IOzone filesystem benchmark. http://www.iozone.org.

[27] ORACLE. What is ZFS? http://docs.oracle.com/cd/E19253-01/819-5461/zfsover-2/.

[28] SCHMIDT, M., HORNUNG, T., LAUSEN, G., AND PINKEL, C. SP^{2}Bench: a SPARQL performance benchmark. In *IEEE ICDE'09* (2009), pp. 222–233.

[29] SELTZER, M., KRINSKY, D., SMITH, K., AND ZHANG, X. The case for application-specific benchmarking. In *USENIX HotOS'99* (1999), pp. 102–107.

[30] SHANNON, C. E. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review 5*, 1 (2001), 3–55.

[31] TARASOV, V., BHANAGE, S., ZADOK, E., AND SELTZER, M. Benchmarking file system benchmarking: It *IS* rocket science. *USENIX HotOS'11* (2011).

[32] TARASOV, V., KUMAR, S., MA, J., HILDEBRAND, D., POVZNER, A., KUENNING, G., AND ZADOK, E. Extracting flexible, replayable models from large block traces. In *USENIX FAST'12* (2012), p. 22.

[33] TARASOV, V., MUDRANKIT, A., BUIK, W., SHILANE, P., KUENNING, G., AND ZADOK, E. Generating realistic datasets for deduplication analysis. In *USENIX ATC'12* (2012), pp. 1–12.

[34] TAY, Y. Data generation for application-specific benchmarking. *VLDB, Challenges and Visions* (2011).

[35] TRAEGER, A., ZADOK, E., JOUKOV, N., AND WRIGHT, C. P. A nine year study of file system and storage benchmarking. *ACM Transactions on Storage (TOS) 4*, 2 (2008), 5.

[36] WANG, L., ZHAN, J., LUO, C., ZHU, Y., YANG, Q., HE, Y., GAO, W., JIA, Z., SHI, Y., ZHANG, S., ZHENG, C., LU, G., ZHAN, K., LI, X., AND QIU, B. BigDataBench: A big data benchmark suite from internet services. In *IEEE HPCA'14* (2014), pp. 488–499.

[37] WIKIPEDIA FOUNDATION. English wikipedia. https://dumps.wikimedia.org/, 2014.

[38] ZHANG, Y., RAJIMWALE, A., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. End-to-end data integrity for file systems: A zfs case study. In *USENIX FAST'10* (2010), pp. 29–42.

[39] ZHENG, Q., CHEN, H., WANG, Y., DUAN, J., AND HUANG, Z. COSBench: A benchmark tool for cloud object storage services. In *IEEE CLOUD'12* (2012), pp. 998–999.

[40] ZHENG, Y., ZHANG, L., XIE, X., AND MA, W.-Y. Mining interesting locations and travel sequences from GPS trajectories. In *WWW'09* (2009), pp. 791–800.

[41] ZIV, J., AND LEMPEL, A. A universal algorithm for sequential data compression. *IEEE Transactions on information theory 23*, 3 (1977), 337–343.

[42] ZIV, J., AND LEMPEL, A. Compression of Individual Sequences via Variable-Rate Coding. *IEEE Transactions on Information Theory 24*, 5 (September 1978), 530–536.

Design Tradeoffs for Data Deduplication Performance in Backup Workloads

Min Fu[†], Dan Feng[†], Yu Hua[†], Xubin He[‡], Zuoning Chen[*], Wen Xia[†], Yucheng Zhang[†], Yujuan Tan[§]
[†]*Wuhan National Lab for Optoelectronics*
School of Computer, Huazhong University of Science and Technology, Wuhan, China
[‡]*Dept. of Electrical and Computer Engineering, Virginia Commonwealth University, VA, USA*
[*]*National Engineering Research Center for Parallel Computer, Beijing, China*
[§]*College of Computer Science, Chongqing University, Chongqing, China*
Corresponding author: dfeng@hust.edu.cn

Abstract

Data deduplication has become a standard component in modern backup systems. In order to understand the fundamental tradeoffs in each of its design choices (such as prefetching and sampling), we disassemble data deduplication into a large N-dimensional parameter space. Each point in the space is of various parameter settings, and performs a tradeoff among backup and restore performance, memory footprint, and storage cost. Existing and potential solutions can be considered as specific points in the space. Then, we propose a general-purpose framework to evaluate various deduplication solutions in the space. Given that no single solution is perfect in all metrics, our goal is to find some reasonable solutions that have sustained backup performance and perform a suitable tradeoff between deduplication ratio, memory footprints, and restore performance. Our findings from extensive experiments using real-world workloads provide a detailed guide to make efficient design decisions according to the desired tradeoff.

1 Introduction

Efficient storage of a huge volume of digital data becomes a big challenge for industry and academia. IDC predicts that there will be 44 ZB of digital data in 2020 [3], which will continue to grow exponentially. Recent work reveals the wide existence of a large amount of duplicate data in storage systems, including primary storage systems [27, 37], secondary backup systems [36], and high-performance data centers [26]. In order to support efficient storage, data deduplication is a widely deployed technique between the underlying storage system and upper applications due to its efficiency and scalability; it becomes increasingly important in large-scale storage systems, especially backup systems.

In backup systems, data deduplication divides a backup stream into non-overlapping variable-sized chunks, and identifies each chunk with a cryptographic digest, such as SHA-1, commonly referred to as a *fingerprint*. Two chunks with identical fingerprints are considered duplicates without requiring a byte-by-byte comparison. The probability of hash collisions is much smaller than that of hardware errors [33], thus it is widely accepted in real-world backup systems. A *fingerprint index* maps fingerprints of the stored chunks to their physical addresses. A duplicate chunk can be identified via checking the existence of its fingerprint in the index. During a backup, the duplicate chunks are eliminated immediately for inline data deduplication. The chunks with unique fingerprints that do not exist in the fingerprint index are aggregated into fixed-sized *containers* (typically 4 MB), which are managed in a log-structure manner [39]. A *recipe* that consists of the fingerprint sequence of the backup is written for future data recovery.

There have been many publications about data deduplication [32]. However, it remains unclear how existing solutions make their design decisions and whether potential solutions can do better. Hence, in the first part of the paper (Section 2), we present a taxonomy to classify existing work using individual design parameters, including **key-value**, **fingerprint prefetching and caching**, **segmenting**, **sampling**, **rewriting**, **restore**, etc. Different from previous surveys [24, 32], our taxonomy is fine-grained with in-depth discussions. We obtain an N-dimensional parameter space, and each point in the space performs a tradeoff among backup and restore performance, memory footprint, and storage cost. Existing solutions are considered as specific points. We figure out how existing solutions choose their points, which allows us to find potentially better solutions. For example, similarity detection in Sparse Indexing [22] and segment prefetching in SiLo [38] are highly complementary.

Although there are some open-source deduplication platforms, such as dmdedup [35], none of them are capable of evaluating the parameter space we discuss. Hence, the second part of our paper (Section 3)

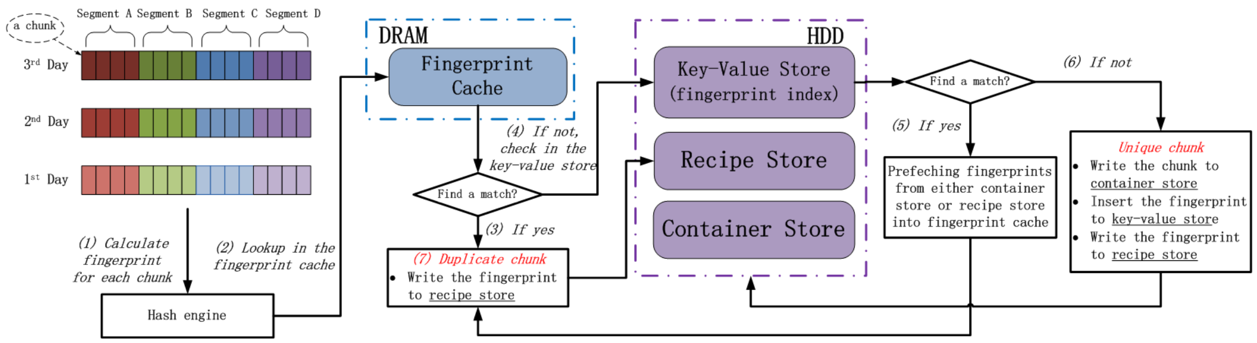

a chunk
Segment A
Segment B
Segment C
Segment D
3rd Day
2nd Day
1st Day
(1) Calculate fingerprint for each chunk
(2) Lookup in the fingerprint cache
Hash engine
DRAM
Fingerprint Cache
(4) If not, check in the key-value store
Find a match?
(3) If yes
(7) Duplicate chunk
Write the fingerprint to recipe store
HDD
Key-Value Store (fingerprint index)
Recipe Store
Container Store
Find a match?
(5) If yes
(6) If not
Prefeching fingerprints from either container store or recipe store into fingerprint cache
Unique chunk
Write the chunk to container store
Insert the fingerprint to key-value store
Write the fingerprint to recipe store

Figure 1: A typical deduplication system and the **Base** deduplication procedure.

presents a general-purpose Deduplication Framework (**DeFrame**)[1], for comprehensive data deduplication evaluation. DeFrame implements the entire parameter space discussed in a modular and extensible fashion; this enables apple-to-apple comparisons among both existing and potential solutions. Our aim is to facilitate finding solutions that provide sustained high backup and restore performance, low memory footprints, and high storage efficiency.

The third part of our paper (Section 4) presents our findings in a large-scale experimental evaluation using real-world long-term workloads. The findings provide a detailed guide to make reasonable decisions according to the desired tradeoff. For example, if high restore performance is required, a rewriting algorithm is required to trade storage efficiency for restore performance. With a rewriting algorithm, the design decisions on the fingerprint index need to be changed. To the best of our knowledge, this is the first work that examines the interplays between fingerprint index and rewriting algorithms.

2 In-line Data Deduplication Space

Figure 1 depicts a typical deduplication system. Generally, we have three components on disks: (1) The *fingerprint index* maps fingerprints of stored chunks to their physical locations. It is used to identify duplicate chunks. (2) The *recipe store* manages recipes that describe the logical fingerprint sequences of concluded backups. A recipe is used to reconstruct a backup stream during restore. (3) The *container store* is a log-structured storage system. While duplicate chunks are eliminated, unique chunks are aggregated into fixed-sized containers. We also have a *fingerprint cache* in DRAM that holds popular fingerprints to boost duplication identification.

Figure 1 also shows the basic deduplication procedure, namely *Base*. At the top left, we have three sample backup streams that correspond to the snapshots of the primary data in three consecutive days. Each backup stream is divided into chunks, and 4 consecutive chunks constitute a *segment* (a segment describes the chunk sequence of

[1]https://github.com/fomy/destor

Parameter list	**Description**
sampling	selecting representative fingerprints
segmenting	splitting the unit of logical locality
segment selection	selecting segments to be prefetched
segment prefetching	exploiting segment-level locality
key-value mapping	multiple logical positions per fingerprint
rewriting algorithm	reducing fragmentation
restore algorithm	designing restore cache

Table 1: The major parameters we discuss.

a piece of data stream; we assume a simplest segmenting approach the this case). Each chunk is processed in the following steps: **(1)** The hash engine calculates the SHA-1 digest for the chunk as its unique identification, namely fingerprint. **(2)** Look up the fingerprint in the in-DRAM fingerprint cache. **(3)** If we find a match, jump to step 7. **(4)** Otherwise, look up the fingerprint in the key-value store. **(5)** If we find a match, invoke a fingerprint prefetching procedure. Jump to step 7. **(6)** Otherwise, the chunk is unique. We write the chunk to the container store, insert the fingerprint to the key-value store, and write the fingerprint to the recipe store. Jump to step 1 to process the next chunk. **(7)** The chunk is a duplicate. We eliminate the chunk and write its fingerprint to the recipe store. Jump to step 1 to process the next chunk.

In the following sections, we (1) propose the fingerprint index subspace (the key component in data deduplication systems) to characterize existing solutions and find potentially better solutions, and (2) discuss the interplays among fingerprint index, rewriting, and restore algorithms. Table 1 lists the major parameters.

2.1 Fingerprint Index

The fingerprint index is a well-recognized performance bottleneck in large-scale deduplication systems [39]. The simplest fingerprint index is only a key-value store [33]. The key is a fingerprint and the value points to the chunk. A duplicate chunk is identified via checking the existence of its fingerprint in the key-value store. Suppose each key-value pair consumes 32 bytes (including a 20-byte fingerprint, an 8-byte container ID, and 4-byte other metadata) and the chunk size is 4 KB on average, indexing 1 TB unique data requires at least an 8 GB-sized key-value store. Putting all fingerprints in DRAM is not cost-efficient. To model the storage cost, we use the unit price from Amazon.com [1]: A Western Digital Blue 1 TB 7200 RPM SATA Hard Drive costs $60, and a Kingston HyperX Blu 8 GB 1600 MHz DDR3 DRAM costs $80. The total storage cost is $140, 57.14% of which is for DRAM.

An HDD-based key-value store suffers from HDD's poor random-access performance, since the fingerprint is completely random in nature. For example, the throughput of Content-Defined Chunking (CDC) is about 400 MB/s under commercial CPUs [11], and hence CDC produces 102,400 chunks per second. Each chunk incurs a lookup request to the key-value store, i.e., 102,400 lookup requests per second. The required throughput is significantly higher than that of HDDs, i.e., 100 IOPS [14]. SSDs support much higher throughput, nearly 75,000 IOPS as venders report [4]. However, SSDs are much more expensive than HDDs and suffer from a performance degradation over time due to reduced over-provisioning space [19].

Due to the incremental nature of backup workloads, the fingerprints of consecutive backups appear in similar sequences [39], which is known as *locality*. In order to reduce the overhead of the key-value store, modern fingerprint indexes leverage locality to prefetch fingerprints, and maintain a *fingerprint cache* to hold the prefetched fingerprints in memory. The fingerprint index hence consists of two submodules: a key-value store and a fingerprint prefetching/caching module. The value instead points to the prefetching unit. According to the use of the key-value store, we classify the fingerprint index into *exact* and *near-exact deduplication*.

- **Exact Deduplication (ED)**: all duplicate chunks are eliminated for highest deduplication ratio (the data size before deduplication divided by the data size after deduplication).

- **Near-exact Deduplication (ND)**: a small number of duplicate chunks are allowed for higher backup performance and lower memory footprint.

According to the fingerprint prefetching policy, we classify the fingerprint index into exploiting *logical* and *physical locality*.

- **Logical Locality (LL)**: the chunk (fingerprint) sequence of a backup stream before deduplication. It is preserved in recipes.

- **Physical Locality (PL)**: the physical layout of chunks (fingerprints), namely the chunk sequence after deduplication. It is preserved in containers.

Figure 2 shows the categories of existing fingerprint indexes. The cross-product of the deduplication and locality variations include **EDPL**, **EDLL**, **NDPL**, and **NDLL**. In the following, we discuss their parameter subspaces and how to choose reasonable parameter settings.

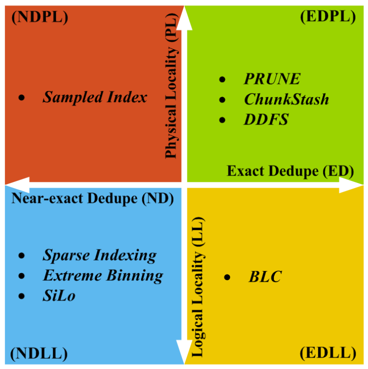


Figure 2: Categories of existing fingerprint indexes. The typical examples include DDFS [39], Sparse Indexing [22], Extreme Binning [10], ChunkStash [14], Sampled Index [18], SiLo [38], PRUNE [28], and BLC [25].

2.1.1 Exact vs. Near-exact Deduplication

The main difference between exact and near-exact deduplication is the use of the key-value store. For exact deduplication, the key-value store has to index the fingerprints of all stored chunks and hence becomes too large to be stored in DRAM. *The fingerprint prefetching/caching module is employed to avoid a large fraction of lookup requests to the key-value store.* Due to the strong locality in backup workloads, the prefetched fingerprints are possibly accessed later. Although the fingerprint index is typically lookup-intensive (most chunks are duplicate in backup workloads), its key-value store is expected not to be lookup-intensive since a large fraction of lookup requests are avoided by the fingerprint prefetching and caching. However, the fragmentation problem discussed in Section 2.1.2 reduces the efficiency of the fingerprint prefetching and caching, making the key-value store become lookup-intensive over time.

For near-exact deduplication, only sampled representative fingerprints, namely *features*, are indexed to downsize the key-value store. With a high sampling ratio (e.g., 128:1), the key-value store is small enough to be completely stored in DRAM. Since only a small fraction of stored chunks are indexed, many duplicate chunks cannot be found in the key-value store. *The fingerprint prefetching/caching module is important to maintain a high deduplication ratio.* Once an indexed duplicate fingerprint is found, many unindexed fingerprints are prefetched to answer following lookup requests. The sampling method is important to the prefetching efficiency, and hence needs to be chosen carefully, which are discussed in Section 2.1.2 and 2.1.3 respectively.

The memory footprint of exact deduplication is related to the key-value store, and proportional to the number of the stored chunks. For example, if we employ Berkeley DB [2] paired with a Bloom filter [12] as the key-value store, 1 byte DRAM per stored chunk is required to maintain a low false positive ratio for the Bloom filter. Suppose S is the average chunk size in KB and M is the DRAM bytes per key, the storage cost per TB stored data includes $\$(\frac{10*M}{S})$ for DRAM and \$60 for HDD. Given $M = 1$ and $S = 4$, the storage cost per TB stored data is \$62.5 (4% for DRAM). In other underlying storage, such as RAID [31], the proportion of DRAM decreases.

The memory footprint of near-exact deduplication depends on the sampling ratio R. Suppose each key-value pair consumes 32 bytes, M is equal to $\frac{32}{R}$. The storage cost per TB stored data includes $\$(\frac{320}{S*R})$ for DRAM and \$60 for HDD. For example, if $R = 128$ and $S = 4$, the storage cost per TB data in near-exact deduplication is \$60.625 (1% for DRAM). However, it is unfair to simply claim that near-exact deduplication saves money, since its stored data includes duplicate chunks. Suppose a 10% loss of deduplication ratio, 1 TB data in near-exact deduplication only stores 90% of 1 TB data in exact deduplication. Hence, near-exact deduplication requires 1.11 TB to store the 1 TB data in exact deduplication. The total storage cost in near-exact deduplication is about \$67.36, higher than exact deduplication. To decrease storage cost, near-exact deduplication has to achieve a high deduplication ratio, no smaller than $(\frac{320}{S*R} + 60)/(\frac{10}{S} + 60)$ of the deduplication ratio in exact deduplication. In our cost model, near-exact deduplication needs to achieve 97% of the deduplication ratio of exact deduplication, which is difficult based on our observations in Section 4.7. Hence, near-exact deduplication generally indicates a cost increase.

2.1.2 Exploiting Physical Locality

The unique chunks (fingerprints) of a backup are aggregated into containers. Due to the incremental nature of backup workloads, the fingerprints in a container are possibly accessed together in subsequent backups [39]. The locality preserved in containers is called *physical locality*. To exploit the physical locality, the value in the key-value store is the container ID and thus the prefetching unit is a container. If a duplicate fingerprint is identified in the key-value store, we obtain a container ID and then read the metadata section (a summary on the fingerprints) of the container into the fingerprint cache. Note that only unique fingerprints are updated with their container IDs in the key-value store.

Although physical locality is an effective approximation of logical locality, the deviation increases over time. For example, old containers have many useless fingerprints for new backups. As a result, the efficiency of the fingerprint prefetching/caching module decreases over time. This problem is known as *fragmentation*, which severely decreases restore performance as reported in recent work [29, 21]. For EDPL, the fragmentation gradually changes the key-value store to be lookup-intensive, and the ever-increasing lookup overhead results in unpredictable backup performance. We cannot know when the fingerprint index will become the performance bottleneck in an aged system.

For NDPL, the sampling method has significant impacts on deduplication ratio. We observe two sampling methods, *uniform* and *random*. The former selects the first fingerprint every R fingerprints in a container, while the latter selects the fingerprints that $mod\ R = 0$ in a container. Although Sampled Index [18] uses the random sampling, we observe the uniform sampling is better. In the random sampling, the missed duplicate fingerprints would not be sampled ($mod\ R \neq 0$) after being written to new containers, making new containers have less features and hence smaller probability of being prefetched. Without this problem, the uniform sampling achieves a significantly higher deduplication ratio.

2.1.3 Exploiting Logical Locality

To exploit logical locality preserved in recipes, each recipe is divided into subsequences called *segments*. A segment describes a fingerprint subsequence of a backup, and maps its fingerprints to container IDs. We identify each segment by a unique ID. The value in the key-value store points to a segment instead of a container, and the segment becomes the prefetching unit. Due to the locality preserved in the segment, the prefetched fingerprints are possibly accessed later. Note that in addition to unique fingerprints, duplicate fingerprints have new segment IDs (unlike physical locality).

For exact deduplication whose key-value store is not in DRAM, it is necessary to access the key-value store as infrequently as possible. Since the Base procedure depicted in Figure 1 follows this principle (only missed-in-cache fingerprints are checked in the key-value store), it is suitable for EDLL. A problem in EDLL is frequently updating the key-value store, since unique and duplicate fingerprints both have new segment IDs. As a result, all fingerprints are updated with their new segment IDs in the key-value store. The extremely high update overhead, which has not been discussed in previous studies, either rapidly wears out an SSD-based key-value store or exhausts the HDD bandwidth. We propose to sample features in segments and only update the segment IDs of unique fingerprints and features in the key-value store. In theory, the sampling would increase lookup overhead, since it leaves many fingerprints along with old segment IDs, leading to a suboptimal prefetching efficiency. However, based on our observations in Section 4.3, the increase of lookup overhead is negligible, making it a reasonable tradeoff. One problem of the sampling optimization is that, after users delete some backups, the

	BLC [25]	Extreme Binning [10]	Sparse Index [22]	SiLo [38]
Exact deduplication	Yes	No	No	No
Segmenting method	FSS	FDS	CDS	FSS & FDS
Sampling method	N/A	Minimum	Random	Minimum
Segment selection	Base	Top-*all*	Top-*k*	Top-1
Segment prefetching	Yes	No	No	Yes
Key-value mapping relationship	1:1	1:1	Varied	1:1

Table 2: Design choices for exploiting logical locality.

fingerprints pointing to stale segments may become unreachable. It would decrease deduplication ratio. A possible solution is to add a column in key-value store to keep container IDs. Only the fingerprints in reclaimed containers are removed in key-value store. The additional storage cost is negligible since EDLL keeps key-value store on disks.

Previous studies on NDLL use similarity detection (described later) instead of the Base procedure [22, 10, 38]. Given the more complicated logic frame and additional in-memory buffer in similarity detection, it is still necessary to make the motivation clear. Based on our observations in Section 4.5, the Base procedure performs well in source code and database datasets (datasets are described in Section 4.1), but underperforms in virtual machine dataset. The major characteristic of virtual machine images is that each image itself contains many duplicate chunks, namely *self-reference*. Self-reference, absent in our source code and database datasets, interferes with the prefetching decision in the Base procedure. A more efficient fingerprint prefetching policy is hence desired in complicated datasets like virtual machine images. As a solution, similarity detection uses a buffer to hold the processing segment, and load the most similar stored segments in order to deduplicate the processing segment. We summarize existing fingerprint indexes exploiting logical locality in Table 2, and discuss similarity detection in a 5-dimensional parameter subspace: segmenting, sampling, segment selection, segment prefetching, and key-value mapping. Note that the following methods of segmenting, sampling, and prefetching are also applicable in the Base procedure.

Segmenting method. The File-Defined Segmenting (FDS) considers each file as a segment [10], which suffers from the greatly varied file size. The Fixed-Sized Segmenting (FSS) aggregates a fixed number (or size) of chunks into a segment [38]. FSS suffers from a shifted content problem similar to the Fixed-Sized Chunking method, since a single chunk insertion/deletion completely changes the segment boundaries. The Content-Defined Segmenting method (CDS) checks the fingerprints in the backup stream [22, 16]. If a chunk's fingerprint matches some predefined rules (e.g., last 10 bits are zeros), the chunk is considered as a segment boundary. CDS is shift-resistant.

Sampling method. It is impractical to calculate the exact similarity of two segments using their all fingerprints. According to Broder [13], the similarity of the two randomly sampled subsets is an unbiased approximation of that of the two complete sets. A segment is considered as a set of fingerprints. A subset of the fingerprints are selected as features since the fingerprints are already random. If two segments share some features, they are considered similar. There are three basic sampling methods: *uniform*, *random*, and *minimum*. The uniform and random sampling methods have been explained in Section 2.1.2. Suppose the sampling ratio is R, the minimum sampling selects the $\frac{segment\ length}{R}$ minimum fingerprints in a segment. Since the distribution of minimum fingerprints is uneven, Aronovich et al. propose to select the fingerprint adjacent to the minimum fingerprint [9]. Only sampled fingerprints are indexed in the key-value store. A smaller R provides more candidates for the segment selection at a cost of increasing the memory footprint. One feature per segment forces a single candidate. The uniform sampling suffers from the problem of content shifting, while the random and minimum sampling are shift-resistant.

Segment selection. After features are sampled in a new segment S, we look up the features in the key-value store to find the IDs of similar segments. There may be many candidates, but not all of them are loaded in the fingerprint cache since too many segment reads hurt backup performance. The similarity-based selection, namely Top-k, selects k most similar segments. Its procedure is as follows: (1) a most similar segment that shares most features with S is selected at a time; (2) the features of the selected segment are eliminated in remaining candidates, to avoid giving scores for features belonging to already selected segments; (3) jump to step 1 to select the next similar segment, until k of segments are selected or we run out of candidates [22]. A more aggressive selection method is to read all similar segments together, namely Top-*all*. A necessary optimization for Top-*all* is to physically aggregate similar segments into a *bin*, and thus a single I/O can fetch all similar segments [10]. A dedicated bin store is hence required. It underperforms if we sample more than 1 feature per segment, since the bins grow big quickly.

Figure 1 illustrates how similar segments arise. Suppose A_1 is the first version of segment A, A_2 is the second version, and so on. Due to the incremental nature of backup workloads, A_3 is possibly similar to its earlier versions, i.e., A_1 and A_2. Such kind of similar segments are *time-oriented*. Generally, reading only the latest version is sufficient. An exceptional case is a segment boundary change, which frequently occurs if fixed-sized segmenting is used. A boundary change may move a part of segment B to segment A, and hence A_3 has two time-oriented similar segments, A_2 and B_2. A larger k is desired to handle these situations. In datasets like virtual machine images, A can be similar with other segments, such as E, due to self-reference. These similar segments are *space-oriented*. Suppose A_2 and E_2 both have 8 features, 6 of them are shared. After deduplicating E_2 at a later time than A_2, 6 of A_2's features are overwritten to be mapped to E_2. E_2 is selected prior to A_2 when deduplicating A_3. A larger k increases the probability of reading A_2. Hence, with many space-oriented segments, a larger k is required to accurately read the time-oriented similar segment at a cost of decreased backup performance.

Segment prefetching. SiLo exploits segment-level locality to prefetch segments [38]. Suppose A_3 is similar to A_2, it is reasonable to expect the next segment B_3 is similar to B_2 that is next to A_2. Fortunately, B_2 is adjacent to A_2 in the recipe, and hence $p-1$ segments following A_2 are prefetched when deduplicating A_3. If more than 1 similar segments are read, segment prefetching can be applied to either all similar segments, as long as $k*p$ segments do not overflow the fingerprint cache, or only the most similar segment.

Segment prefetching has at least two advantages: 1) *Reducing lookup overhead*. Prefetching B_2 together with A_2, while deduplicating A_3, avoids the additional I/O of reading B_2 for the following B_3. 2) *Improving deduplication ratio*. Assuming the similarity detection fails for B_3 (i.e., B_2 is not found due to its features being changed), the previous segment prefetching caused by A_3, whose similarity detection succeeds (i.e., A_2 is read for A_3 and B_2 is prefetched together), offers a deduplication opportunity for B_3. Segment prefetching also alleviates the problem caused by segment boundary change. In the case of two time-oriented similar segments, A_2 and B_2 would be prefetched for A_3 even if $k=1$. Segment prefetching relies on storing segments in a logical sequence being incompatible with Top-*all*.

Key-value mapping relationship. The key-value store maps features to stored segments. Since a feature can belong to different segments (hence multiple logical positions), the key can be mapped to multiple segment IDs. As a result, the value becomes a FIFO queue of segment IDs, where v is the queue size. For NDLL, maintaining the queues has low performance overhead since the key-value store is in DRAM. A larger v provides more similar segment candidates at a cost of a higher memory footprint. It is useful in the following cases: 1) *Self-reference is common*. A larger v alleviates the above problem of feature overwrites caused by space-oriented similar segments. 2) *The corrupted primary data is restored to an earlier version rather than the latest version (rollback)*. For example, if A_2 has some errors, we roll back to A_1 and thus A_3 derives from A_1 rather than A_2. In this case, A_1 is a better candidate than A_2 for deduplicating A_3, however features of A_1 have been overwritten by A_2. A larger v avoids this problem.

2.2 Rewriting and Restore Algorithms

Since the fragmentation decreases the restore performance in aged systems, the rewriting algorithm, an emerging dimension in the parameter space, was proposed to allow sustained high restore performance [20, 21, 17]. It identifies fragmented duplicate chunks and rewrites them to new containers. Even though near-exact deduplication trades deduplication ratio for restore performance, our observations in Section 4.6 show that the rewriting algorithm is a more efficient tradeoff. However, the rewriting algorithm's interplay with the fingerprint index has not yet been discussed.

We are mainly concerned about two questions. (1) *How does the rewriting algorithm reduce the ever-increasing lookup overhead of EDPL?* Since the rewriting algorithm reduces the fragmentation, EDPL is improved because of better physical locality. Our observations in Section 4.6 show that, via an efficient rewriting algorithm, the lookup overhead of EDPL no longer increases over time. EDPL then has sustained backup performance. (2) *Does the fingerprint index return the latest container ID when a recently rewritten chunk is checked?* Each rewritten chunk would have a new container ID. If the old container ID is returned when that chunk is recirculated, then another rewrite could occur. Based on our observations, this problem is more pronounced and significant in EDLL than EDPL. An intuitive explanation is that, due to our sampling optimization mentioned in Section 2.1.3, an old segment containing obsolete container IDs is read for deduplication. As a result, EDPL becomes better than EDLL due to its higher deduplication ratio.

While the rewriting algorithm determines the chunk placement, an efficient restore algorithm leverages the placement to gain better restore performance with a limited memory footprint. There have been three restore algorithms: the basic LRU cache, the forward assembly area (ASM) [21], and the optimal cache (OPT) [17]. In all of them, a container serves as the prefetching unit during a restore to leverage locality. Their major difference is that while LRU and OPT use container-level replacement, ASM uses chunk-level replacement. We observe these algorithms' performances under different placements in Section 4.6. If the fragmentation is dominant, ASM is more efficient. The reason is that LRU

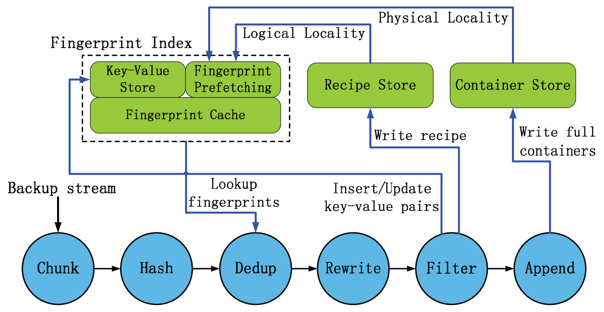


Figure 3: The architecture and backup pipeline.

and OPT hold many useless chunks that are not restored in DRAM due to their container-level replacement. On the other hand, if an efficient rewriting algorithm has reduced the fragmentation, the container-level replacement is improved due to better physical locality and OPT performs best due to its accurate cache replacement.

3 The DeFrame Framework

The N-dimensional parameter space discussed in Section 2 provides a large amount of design choices, but there is no platform to evaluate these choices as far as we know. In this section, we present *DeFrame* as a general-purpose chunk-level deduplication framework to facilitate exploring alternatives. In DeFrame, existing and potential solutions are considered as specific points in the N-dimensional parameter space. We implement DeFrame using C and pthreads on 64-bit Linux.

3.1 Architecture

As shown in Figure 3, DeFrame consists of three submodules discussed in Section 2. In the *container store*, each container is identified by a globally unique ID. The container is the prefetching unit for exploiting physical locality. Each container includes a metadata section that summarizes the fingerprints of all chunks in the container. We can fetch an entire container or only its metadata section via a container ID.

The *recipe store* manages recipes of all finished backups. In recipes, the associated container IDs are stored along with fingerprints so as to restore a backup without the need to consult the fingerprint index. We add some indicators of segment boundaries in each recipe to facilitate reading a segment that is the prefetching unit for exploiting logical locality. Each segment is identified by a globally unique ID. For example, an ID can consist of a 2-byte pointer to a recipe, a 4-byte offset in the recipe, and a 2-byte segment size that indicates how many chunks are in the segment.

The *fingerprint index* consists of a key-value store and a fingerprint prefetching/caching module. Two kinds of key-value stores are currently supported: an in-DRAM hash table and a MySQL database [6] paired with a Bloom filter. Since we implement a virtual layer upon the key-value store, it is easy to add a new key-value store.

3.2 Backup Pipeline

As shown in Figure 3, we divide the workflow of data deduplication into six phases: *Chunk*, *Hash*, *Dedup*, *Rewrite*, *Filter*, and *Append*. (1) The **Chunk** phase divides the backup stream into chunks. We have implemented Fixed-Sized Chunking and Content-Defined Chunking (CDC). (2) The **Hash** phase calculates a SHA-1 digest for each chunk as the fingerprint. (3) The **Dedup** phase aggregates chunks into segments, and identifies duplicate chunks via consulting the fingerprint index. A duplicate chunk is marked and obtains the container ID of its stored copy. The created segments are the prefetching units of logical locality, and the batch process units for physical locality. We have implemented the Base, Top-k, and *Mix* procedures (first Top-k then Base). (4) The **Rewrite** phase identifies fragmented duplicate chunks, and rewrites them to improve restore performance. It is a tradeoff between deduplication ratio and restore performance. We have implemented four rewriting algorithms, including CFL-SD [30], CBR [20], Capping [21], and HAR [17]. Each fragmented chunk is marked. (5) The **Filter** phase handles chunks according to their marks. Unique and fragmented chunks are added to the container buffer. Once the container buffer is full, it is pushed to the next phase. The recipe store and key-value store are updated. (6) The **Append** phase writes full containers to the container store.

We pipeline the phases via pthreads to leverage multicore architecture. The dedup, rewrite, and filter phases are separated for modularity: we can implement a new rewriting algorithm without the need to modify the fingerprint index, and vice versa.

Segmenting and Sampling. The segmenting method is called in the dedup phase, and the sampling method is called for each segment either in the dedup phase for the similarity detection, or in the filter phase for the Base procedure. All segmenting and sampling methods mentioned in Section 2 have been implemented. Content-defined segmenting is implemented via checking the last n bits of a fingerprint. If all the bits are zero, the fingerprint (chunk) is considered to be the beginning of a new segment, thus generating an average segment size of 2^n chunks. To select the first fingerprint of a content-defined segment as a feature, the random sampling also checks the last $\log_2 R$ ($< n$) bits.

3.3 Restore Pipeline

The restore pipeline in DeFrame consists of three phases: *Reading Recipe*, *Reading Chunks*, and *Writing Chunks*. **(1) Reading Recipe**. The required backup recipe is opened for restore. The fingerprints are read and issued one by one to the next step. **(2) Reading Chunks**. Each fingerprint incurs a chunk read request. The container

Dataset name	Kernel	VMDK	RDB
Total size	104 GB	1.89 TB	1.12 TB
# of versions	258	127	212
Deduplication ratio	45.28	27.36	39.1
Avg. chunk size	5.29 KB	5.25 KB	4.5 KB
Self-reference	$< 1\%$	15-20%	0
Fragmentation	Severe	Moderate	Severe

Table 3: The characteristics of datasets.

is read from the container store to satisfy the request. A chunk cache is maintained to hold popular chunks in memory. We have implemented three kinds of restore algorithms, including the basic LRU cache, the optimal cache [17], and the rolling forward assembly area [21]. Given a chunk placement determined by the rewriting algorithm, a good restore algorithm boosts the restore procedure with a limited memory footprint. The required chunks are issued one by one to the next phase. **(3) Writing Chunks**. Using the received chunks, files are reconstructed in the local file system.

3.4 Garbage Collection

After users delete expired backups, chunks become invalid (not referenced by any backup) and must be reclaimed. There are a number of possible techniques for *garbage collection* (GC), such as reference counting [34] and mark-and-sweep [18]. Extending the DeFrame taxonomy to allow comparison of GC techniques is beyond the scope of this work; currently, DeFrame employs the History-Aware Rewriting (HAR) algorithm and Container-Marker Algorithm (CMA) proposed in [17]. HAR rewrites fragmented valid chunks to new containers during backups, and CMA reclaims old containers that are no longer referenced.

4 Evaluation

In this section, we evaluate the parameter space to find reasonable solutions that perform suitable tradeoffs.

4.1 Experimental Setup

We use three real-world datasets as shown in Table 3. Kernel is downloaded from the web [5]. It consists of 258 versions of unpacked Linux kernel source code. VMDK is from a virtual machine with Ubuntu 12.04. We compiled the source code, patched the system, and ran an HTTP server on the virtual machine. VMDK has many self-references; it also has less fragmentation from its fewer versions and random updates. RDB consists of Redis database [7] snapshots. The database has 5 million records, 5 GB in space and an on average 1% update ratio. We disable the default rdbcompression option.

All datasets are divided into variable-sized chunks via CDC. We use the content-defined segmenting with an average segment size of 1024 chunks by default. The container size is 4 MB, which is close to the average size of segments. The default fingerprint cache has 1024 slots to hold prefetching units, being either containers or segments. Hence, the cache can hold 1 million fingerprints, which is relatively large for our datasets.

4.2 Metrics and Our Goal

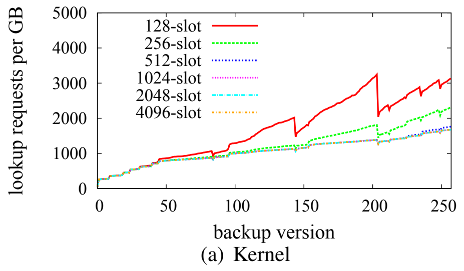
Our evaluations are in terms of quantitative metrics listed as follow. (1) *Deduplication ratio*: the original backup data size divided by the size of stored data. It indicates how efficiently data deduplication eliminates duplicates, being an important factor in the storage cost. (2) *Memory footprint*: the runtime DRAM consumption. A low memory footprint is always preferred due to DRAM's high unit price and energy consumption. (3) *Storage cost*: the cost for storing chunks and the fingerprint index, including memory footprint. We ignore the cost for storing recipes, since it is constant. (4) *Lookup requests per GB*: the number of required lookup requests to the key-value store to deduplicate 1 GB of data, most of which are random reads. (5) *Update requests per GB*: the number of required update requests to the key-value store to deduplicate 1 GB of data. A higher lookup/update overhead degrades the backup performance. Lookup requests to unique fingerprints are eliminated since most of them are expected to be answered by the in-memory Bloom filter. (6) *Restore speed*: 1 divided by mean containers read per MB of restored data [21]. It is used to evaluate restore performance, where a higher value is better. Since the container size is 4 MB, 4 units of restore speed translate to the maximum storage bandwidth.

It is practically impossible to find a solution that performs the best in all metrics. Our goal is to find some reasonable solutions with the following properties: (1) sustained, high backup performance as the top priority; (2) reasonable tradeoffs in the remaining metrics.

4.3 Exact Deduplication

Previous studies [39, 14] of EDPL fail to have an insight of the impacts of the fragmentation on the backup performance, since their datasets are short-term. Figure 4 shows the ever-increasing lookup overhead. We observe 6.5-12.0× and 5.1-114.4× increases in Kernel and RDB respectively under different fingerprint cache sizes. A larger cache cannot address the fragmentation problem; a 4096-slot cache performs as poor as the default 1024-slot cache. A 128-slot cache results in a 114.4× increase in RDB, which indicates an insufficient cache can result in unexpectedly poor performance. This causes complications in practice due to the difficulty in predicting how much memory is required to avoid unexpected performance degradations. Furthermore, even with a large cache, the lookup overhead still increases over time.

Before comparing EDLL to EDPL, we need to determine the best segmenting and sampling methods for EDLL. Figure 5(a) shows the lookup/update overheads



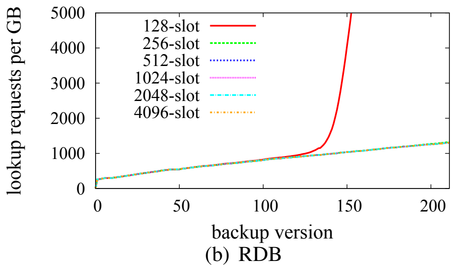
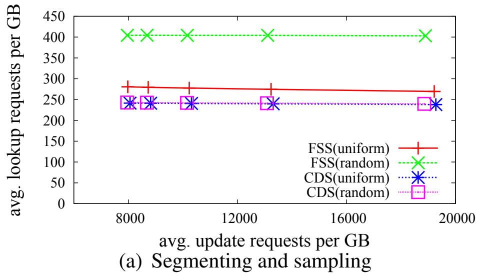


Figure 4: The ever-increasing lookup overhead of EDPL in Kernel and RDB under various fingerprint cache sizes.



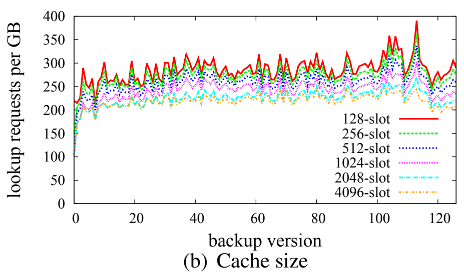
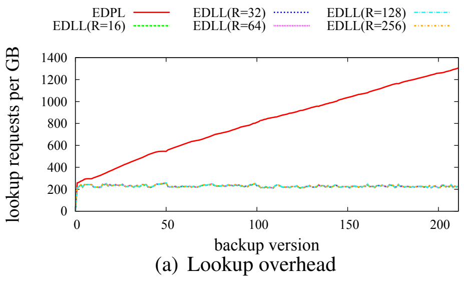


Figure 5: Impacts of varying segmenting, sampling, and cache size on EDLL in VMDK. **(a)** FSS is Fixed-Sized Segmenting and CDS is Content-Defined Segmenting. Points in a line are of different sampling ratios, which are 256, 128, 64, 32, and 16 from left to right. **(b)** EDLL is of CDS and a 256:1 random sampling ratio.



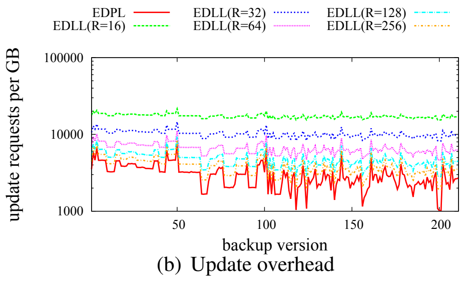


Figure 6: Comparisons between EDPL and EDLL in terms of lookup and update overheads. $R = 256$ indicates a sampling ratio of 256:1. Results come from RDB.

of EDLL under different segmenting and sampling methods in VMDK. Similar results are observed in Kernel and RDB. Increasing the sampling ratio shows an efficient tradeoff: a significantly lower update overhead at a negligible cost of a higher lookup overhead. The fixed-sized segmenting paired with the random sampling performs worst. This is because it cannot sample the first fingerprint in a segment, which is important for the Base procedure. The other three combinations are more efficient since they sample the first fingerprint (the random sampling performs well in the content-defined segmenting due to our optimization in Section 3.2). The content-defined segmenting is better than the fixed-sized segmenting due to its shift-resistance. Figure 5(b) shows the lookup overheads in VMDK under different cache sizes. We do not observe an ever-increasing trend of lookup overhead in EDLL. A 128-slot cache results in additional I/O (17% more than the default) due to the space-oriented similar segments in VMDK. Kernel and RDB (not shown in the figure) do not cause this problem because they have no self-reference.

Figure 6 compares EDPL and EDLL in terms of lookup and update overheads. EDLL uses the content-defined segmenting and random sampling. Results in Kernel and VMDK are not shown, because they have similar results to RDB. While EDPL suffers from the ever-increasing lookup overhead, EDLL has a much lower and sustained lookup overhead (3.6× lower than EDPL on average). With a 256:1 sampling ratio, EDLL has 1.29× higher update overhead since it updates sampled duplicate fingerprints with their new segment IDs. Note that lookup requests are completely random, and update requests can be optimized to sequential writes via a log-structured key-value store, which is a popular design [8, 23, 15]. Overall, if the highest deduplication ratio is required, EDLL is a better choice due to its sustained high backup performance.

Finding (1): While the fragmentation results in an ever-increasing lookup overhead in EDPL, EDLL achieves sustained performance. The sampling optimization performs an efficient tradeoff in EDLL.

4.4 Near-exact Deduplication exploiting Physical Locality

NDPL is simple and easy to implement. Figure 7(a) shows how to choose an appropriate sampling method for NDPL. We only show the results from VMDK, which are similar to the results from Kernel and RDB. The uniform sampling achieves significantly higher deduplication ratio than the random sampling. The reason has been discussed in Section 2.1.2; for the random sampling, the missed duplicate fingerprints are definitely not sampled, making new containers have less features and hence smaller probability of being prefetched. The sampling ratio is a tradeoff between memory footprint and deduplication ratio: a higher sampling ratio indicates a lower memory footprint at a cost of a decreased deduplication ratio. Figure 7(b) shows that NDPL is surprisingly resistent to small cache sizes: a 64-slot cache results in only an 8% decrease of the deduplication ratio than the default in RDB. Also observed (not shown in the figure) are 24-93% additional I/O, which come from prefetching fingerprints. Compared to EDPL, NDPL has better backup performance because of its in-memory key-value store at a cost of decreasing deduplication ratio.

Finding (2): In NDPL, the uniform sampling is better than the random sampling. The fingerprint cache has minimal impacts on deduplication ratio.

4.5 Near-exact Deduplication exploiting Logical Locality

Figure 8(a) compares the Base procedure (see Figure 1) to the simplest similarity detection Top-1, which helps to choose appropriate sampling method. The content-defined segmenting is used due to its advantage shown in EDLL. In the Base procedure, the random sampling achieves comparable deduplication ratio using less memory than the uniform sampling. NDLL is expected to outperform NDPL in terms of deduplication ratio since NDLL does not suffer from fragmentation. However, we surprisingly observe that, while NDLL does better in Kernel and RDB as expected, NDPL is better in VMDK (shown in Figure 7(b) and 8(b)). The reason is that self-reference is common in VMDK. The fingerprint prefetching is misguided by space-oriented similar segments as discussed in Section 2.1.3. Moreover, the fingerprint cache contains many duplicate fingerprints that reduce the effective cache size, therefore a 4096-slot cache improves deduplication ratio by 7.5%. NDPL does not have this problem since its prefetching unit (i.e., container) is after-deduplication. A 64-slot cache results in 23% additional I/Os in VMDK (not shown in the figure), but has no side-effect in Kernel and RDB.

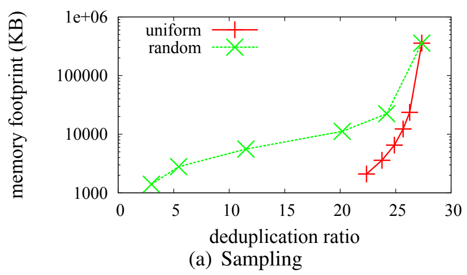
In the Top-1 procedure, only the most similar segment is read. The minimum sampling is slightly better than the random sampling. The Top-1 procedure is worse than the Base procedure. The reason is two-fold as discussed in Section 2.1.3: (1) a segment boundary change results in more time-oriented similar segments; (2) self-reference results in many space-oriented similar segments.

Finding (3): The Base procedure underperforms in NDLL if self-reference is common. Reading a single most similar segment is insufficient due to self-reference and segment boundary changes.

We further examine the remaining NDLL subspace: segment selection (s), segment prefetching (p), and mapping relationship (v). Figure 9 shows the impacts of varying the three parameters on deduplication ratio (lookup overheads are omitted due to space limits). On the X-axis, we have parameters in the format (s, p, v). The s indicates the segment selection method, being either Base or Top-k. The p indicates the number of prefetched segments plus the selected segment. We apply segment prefetching to all similar segments selected. The v indicates the maximum number of segments that a feature refers to. The random sampling is used, with a sampling ratio of 128. For convenience, we use NDLL(s, p, v) to represent a point in the space.

A larger v results in a higher lookup overhead when $k > 1$, since it provides more similar segment candidates. We observe that increasing v is not cost-effective in Kernel which lacks self-reference, since it increases lookup overhead without an improvement of deduplication ratio. However, in RDB which also lacks of self-reference, NDLL(Top-1,1,2) achieves better deduplication ratio than NDLL(Top-1,1,1) due to the rollbacks in RDB. A larger v is helpful to improve deduplication ratio in VMDK where self-reference is common. For example, NDLL(Top-1,1,2) achieves 1.31× higher deduplication ratio than NDLL(Top-1,1,1) without an increase of lookup overhead.

The segment prefetching is efficient for increasing deduplication ratio and decreasing lookup overhead. As the parameter p increases from 1 to 4 in the Base procedure, the deduplication ratios increase by 1.06×, 1.04×, and 1.39× in Kernel, RDB, and VMDK respectively, while the lookup overheads decrease by 3.81×, 3.99×, and 3.47×. The Base procedure is sufficient to achieve a high deduplication ratio in Kernel and RDB that lack self-reference. Given its simple logical frame, the Base procedure is a reasonable choice if self-reference is rare.



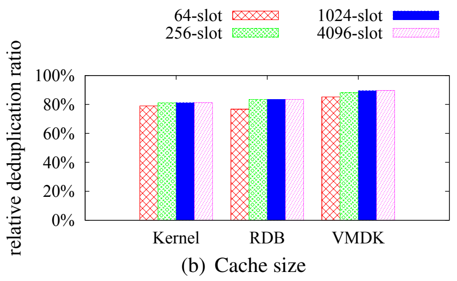
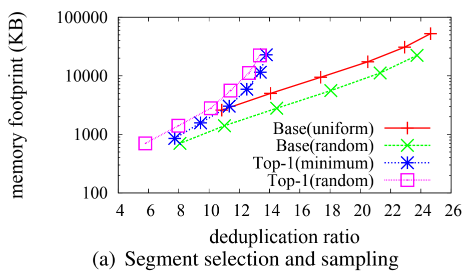


Figure 7: Impacts of varying sampling method and cache size on NDPL. **(a)** Points in each line are of different sampling ratios, which are 256, 128, 64, 32, 16, and 1 from left to right. **(b)** NDPL uses a 128:1 uniform sampling ratio. The Y-axis shows the relative deduplication ratio to exact deduplication.



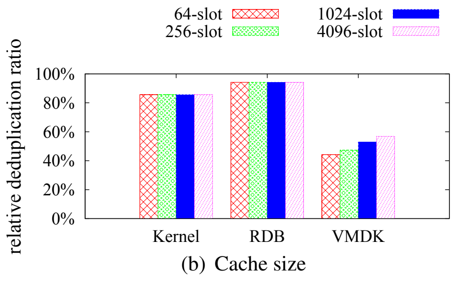


Figure 8: **(a)** Comparisons of Base and Top-1 in VMDK. Points in each line are of different sampling ratios, which are 512, 256, 128, 64, 32, and 16 from left to right. **(b)** NDLL is of the Base procedure and a 128:1 random sampling. The Y-axis shows the relative deduplication ratio to exact deduplication.

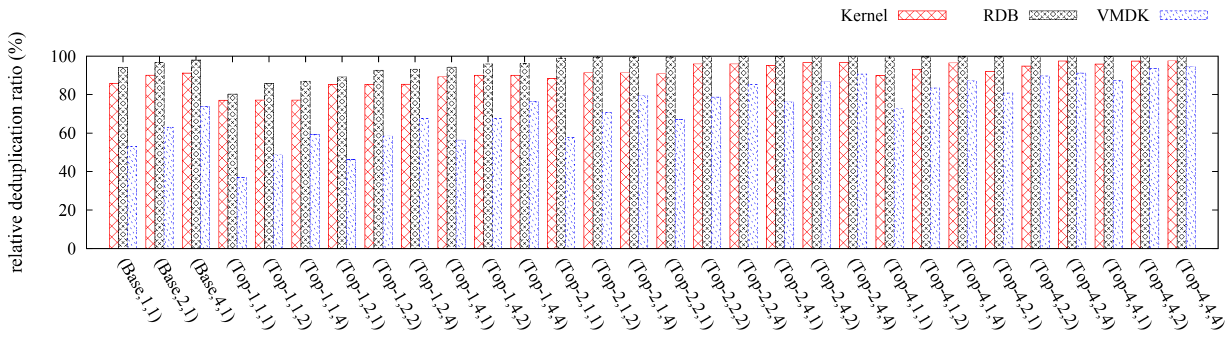


Figure 9: Impacts of the segment selection, segment prefetching, and mapping relationship on deduplication ratio. The deduplication ratios are relative to those of exact deduplication.

However, the Base procedure only achieves a 73.74% deduplication ratio of exact deduplication in VMDK where self-reference is common.

Finding (4): If self-reference is rare, the Base procedure is sufficient for a high deduplication ratio.

In more complicated environments like virtual machine storage, the Top-*k* procedure is desired. A higher *k* indicates a higher deduplication ratio at a cost of a higher lookup overhead. As *k* increases from NDLL(Top-1,1,1) to NDLL(Top-4,1,1), the deduplication ratios increase by 1.17×, 1.24×, and 1.97× in Kernel, RDB, and VMDK respectively, at a cost of 1.15×, 1.01×, and 1.56× more segment reads. Note that Top-4 outperforms Base in terms of deduplication ratio in all datasets. Varying *k* has fewer impacts in Kernel and RDB, since they have fewer space-oriented similar segments and hence fewer candidates. The segment prefetching is a great complement to the Top-*k* procedure, since it amortizes the additional lookup overhead caused by increasing *k*. NDLL(Top-4,4,1) reduces the lookup overheads of NDLL(Top-4,1,1) by 2.79×, 3.97×, and 2.07× in Kernel, RDB, and VMDK respectively. It also improves deduplication ratio by a factor of 1.2× in VMDK. NDLL(Top-4,4,1) achieves a 95.83%, 99.65%, and 87.20% deduplication ratio of exact deduplication, significantly higher than NDPL.

Finding (5): If self-reference is common, the similarity detection is required. The segmenting prefetching is a great complement to Top-k.

	Dataset	*EDPL*	*NDPL-128*	*NDPL-256*	*NDPL-512*	*HAR+EDPL*
	Kernel	45.35	36.86	33.52	30.66	31.26
Deduplication ratio	RDB	39.10	32.64	29.31	25.86	28.28
	VMDK	27.36	24.50	23.15	21.46	24.90
	Kernel	0.50	0.92	1.04	1.19	2.60
Restore speed	RDB	0.50	0.82	0.87	0.95	2.26
	VMDK	1.39	2.49	2.62	2.74	2.80

Table 4: Comparisons between near-exact deduplication and rewriting in terms of restore speed and deduplication ratio. NDPL-256 indicates NDPL of a 256:1 uniform sampling ratio. HAR uses EDPL as the fingerprint index. The restore cache contains 128 containers for Kernel, and 1024 containers for RDB and VMDK.

4.6 Rewriting Algorithm and its Interplay

Fragmentation decreases restore performance significantly in aged systems. The rewriting algorithm is proposed to trade deduplication ratio for restore performance. To motivate the rewriting algorithm, Table 4 compares near-exact deduplication to a rewriting algorithm, History-Aware Rewriting algorithm (HAR) [17]. We choose HAR due to its accuracy in identifying fragmentation. As the baseline, EDPL has best deduplication ratio and hence worst restore performance. NDPL shows its ability of improving restore performance, however not as well as HAR. Taking RDB as an example, NDPL of a 512:1 uniform sampling ratio trades 33.88% deduplication ratio for only 1.18× improvement in restore speed, while HAR trades 27.69% for 2.8× improvement.

We now answer the questions in Section 2.2: (1) How does the rewriting algorithm improve EDPL in terms of lookup overhead? (2) How does fingerprint index affect the rewriting algorithm? Figure 10(a) shows how HAR improves EDPL. We observe that HAR successfully stops the ever-increasing trend of lookup overhead in EDPL. Although EDPL still has a higher lookup overhead than EDLL, it is not a big deal because a predictable and sustained performance is the main concern. Moreover, HAR has no impact on EDLL, since EDLL does not exploit physical locality that HAR improves. The periodic spikes are because of major updates in Linux kernel, such as from 3.1 to 3.2. These result in many new chunks, which reduce logical locality. Figure 10(b) shows how fingerprint index affects HAR. EDPL outperforms EDLL in terms of deduplication ratio in all datasets. As explained in Section 2.2, EDLL could return an obsolete container ID if an old segment is read, and hence a recently rewritten chunk would be rewritten again. Overall, with an efficient rewriting algorithm, EDPL is a better choice than EDLL due to its higher deduplication ratio and sustained performance.

Finding (6): The rewriting algorithm helps EDPL to achieve sustained backup performance. With a rewriting algorithm, EDPL is better due to its higher deduplication ratio than other index schemes.

We further examine three restore algorithms: the LRU cache, the forward assembly area (ASM) [21], and the optimal cache (OPT) [17]. Figure 11 shows the efficiencies of these restore algorithms with and without HAR in Kernel and VMDK. Because the restore algorithm only matters under limited memory, the DRAM used is smaller than Table 4, 32-container-sized in Kernel and 256-container-sized in VMDK. If no rewriting algorithm is used, the restore performance of EDPL decreases over time due to the fragmentation. ASM has better performance than LRU and OPT, since it never holds useless chunks in memory. If HAR is used, EDPL has sustained high restore performance since the fragmentation has been reduced. OPT is best in this case due to its efficient cache replacement.

Finding (7): Without rewriting, the forward assembly area is recommended; but with an efficient rewriting algorithm, the optimal cache is better.

4.7 Storage Cost

As discussed in Section 2, indexing 1 TB unique data of 4 KB chunks in DRAM, called *baseline*, costs \$140, 57.14% of which is for DRAM. The cost is even higher if considering the high energy consumption of DRAM. The baseline storage costs are \$0.23, \$3.11, and \$7.55 in Kernel, RDB, and VMDK respectively.

To reduce the storage cost, we either use HDD instead of DRAM for exact deduplication or index a part of fingerprints in DRAM for near-exact deduplication. Table 5 shows the relative storage costs to the baseline in each dataset. EDPL and EDLL have the identical storage cost, since they have the same deduplication ratio and key-value store. We assume that the key-value store in EDPL and EDLL is a database paired with a Bloom filter, hence 1 byte DRAM per stored chunk for a low false positive ratio. EDPL and EDLL reduce the storage cost by a factor of around 1.75. The fraction of the DRAM cost is 2.27-2.50%.

Near-exact deduplication of a high sampling ratio further reduces the DRAM cost, at a cost of decreasing deduplication ratio. As discussed in Section 2.1.1, near-exact deduplication with a 128:1 sampling ratio and 4 KB chunk size needs to achieve 97% of deduplication ratio of exact deduplication to avoid a cost increase. To evaluate this tradeoff, we observe the storage costs of NDPL and NDLL under various sampling ratios. NDPL uses the

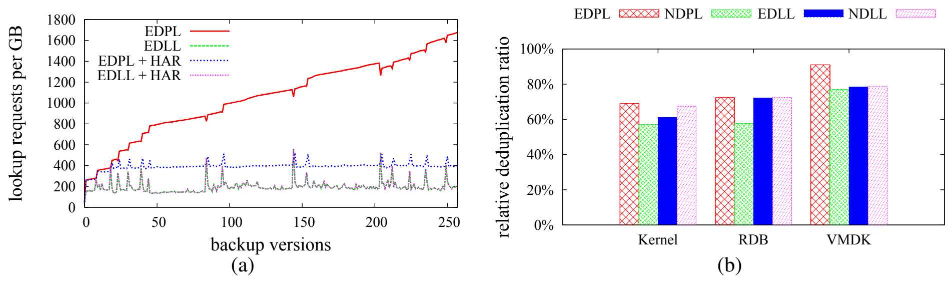


Figure 10: Interplays between fingerprint index and rewriting algorithm (i.e., HAR). **(a)** How does HAR improve EDPL in terms of lookup overhead in Kernel? **(b)** How does fingerprint index affect HAR? The Y-axis shows the relative deduplication ratio to that of exact deduplication without rewriting.

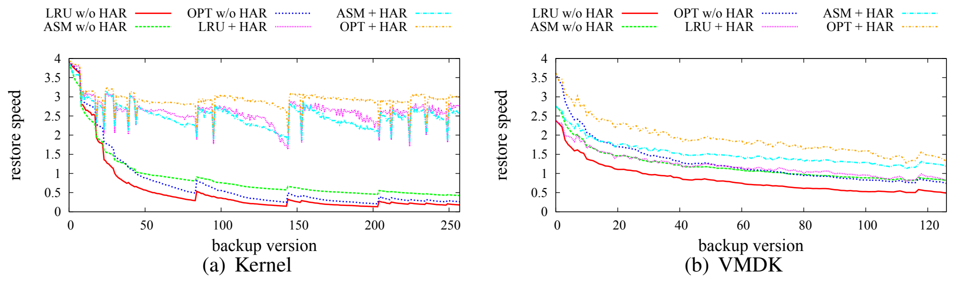


Figure 11: Interplays between the rewriting and restore algorithms. EDPL is used as the fingerprint index.

Dataset	*Fraction*	*EDPL/EDLL*	*NDPL-64*	*NDPL-128*	*NDPL-256*	*NDLL-64*	*NDLL-128*	*NDLL-256*
Kernel	DRAM	1.33%	0.83%	0.49%	0.31%	0.66%	0.34%	0.16%
	HDD	57.34%	65.01%	70.56%	77.58%	59.03%	59.83%	60.23%
	Total	58.67%	65.84%	71.04%	77.89%	59.69%	60.17%	60.39%
RDB	DRAM	1.40%	0.83%	0.48%	0.31%	0.70%	0.35%	0.17%
	HDD	55.15%	61.25%	66.08%	73.58%	55.27%	55.34%	55.65%
	Total	56.55%	62.07%	66.56%	73.89%	55.97%	55.69%	55.82%
VMDK	DRAM	1.41%	0.82%	0.45%	0.27%	0.71%	0.35%	0.18%
	HDD	54.86%	60.32%	63.16%	67.10%	59.79%	62.92%	71.24%
	Total	56.27%	61.14%	63.61%	67.36%	60.49%	63.27%	71.42%

Table 5: The storage costs relative to the baseline which indexes all fingerprints in DRAM. NDPL-128 is NDPL of a 128:1 uniform sampling ratio.

uniform sampling, and NDLL is of the parameter (Top-4,4,1). As shown in Table 5, NDPL increases the storage cost in all datasets; NDLL increases the storage cost in most cases, except in RDB.

Finding (8): Although near-exact deduplication reduces the DRAM cost, it cannot reduce the total storage cost.

5 Conclusions

In this paper, we discuss the parameter space of data deduplication in detail, and we present a general-purpose framework called DeFrame for evaluation. DeFrame can efficiently find reasonable solutions to explore tradeoffs among backup and restore performance, memory footprints, and storage costs. Our findings, from a large-scale evaluation using three real-world long-term workloads, provide a detailed guide to make efficient design decisions for deduplication systems.

It is impossible to have a solution that performs the best in all metrics, To achieve the lowest storage cost, Exact Deduplication exploiting Logical Locality (EDLL) is preferred due to its highest deduplication ratio and sustained high backup performance. To achieve the lowest memory footprint, Near-exact Deduplication is recommended: either exploiting Physical Locality (NDPL) for its simpleness, or exploiting Logical Locality (NDLL) for better deduplication ratio. To achieve a sustained high restore performance, Exact Deduplication exploiting Physical Locality (EDPL) combined with a rewriting algorithm would be a better choice.

Acknowledgments

We are grateful to our shepherd Fred Douglis and the anonymous reviewers for their insightful feedback. We appreciate Benjamin Young for his work to proofread the final version. The work was partly supported by

National Basic Research 973 Program of China under Grant No. 2011CB302301; NSFC No. 61025008, 61173043, 61232004, and 6140050892; 863 Project 2013AA013203; Fundamental Research Funds for the Central Universities, HUST, under Grant No. 2014QN-RC019. The work conducted at VCU was partially sponsored by US National Science Foundation under Grants CNS-1320349 and CNS-1218960. The work was also supported by Key Laboratory of Information Storage System, Ministry of Education, China.

References

[1] Amazon. http://www.amazon.com, 2014.

[2] Berkeley DB. http://www.oracle.com/us/products/database/berkeley-db/overview/index.htm, 2014.

[3] The digital universe of opportunities. http://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm, 2014.

[4] Intel solid-state drive dc s3700 series. http://www.intel.com/content/www/us/en/solid-state-drives/solid-state-drives-dc-s3700-series.html, 2014.

[5] Linux kernel. http://www.kernel.org/, 2014.

[6] MySQL. http://www.mysql.com/, 2014.

[7] Redis. http://redis.io/, 2014.

[8] ANAND, A., MUTHUKRISHNAN, C., KAPPES, S., AKELLA, A., AND NATH, S. Cheap and large CAMs for high performance data-intensive networked systems. In *Proc. USENIX NSDI, 2010*.

[9] ARONOVICH, L., ASHER, R., BACHMAT, E., BITNER, H., HIRSCH, M., AND KLEIN, S. T. The design of a similarity based deduplication system. In *Proc. ACM SYSTOR, 2009*.

[10] BHAGWAT, D., ESHGHI, K., LONG, D. D., AND LILLIBRIDGE, M. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Proc. IEEE MASCOTS, 2009*.

[11] BHATOTIA, P., RODRIGUES, R., AND VERMA, A. Shredder : Gpu-accelerated incremental storage and computation. In *Proc. USENIX FAST, 2012*.

[12] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM 13*, 7 (July 1970), 422–426.

[13] BRODER, A. On the resemblance and containment of documents. In *Compression and Complexity of Sequences 1997. Proceedings* (Jun 1997), pp. 21–29.

[14] DEBNATH, B., SENGUPTA, S., AND LI, J. ChunkStash: speeding up inline storage deduplication using flash memory. In *Proc. USENIX ATC, 2010*.

[15] DEBNATH, B., SENGUPTA, S., AND LI, J. SkimpyStash: Ram space skimpy key-value store on flash-based storage. In *Proc. ACM SIGMOD, 2011*.

[16] DONG, W., DOUGLIS, F., LI, K., PATTERSON, H., REDDY, S., AND SHILANE, P. Tradeoffs in scalable data routing for deduplication clusters. In *Proc. USENIX FAST, 2011*.

[17] FU, M., FENG, D., HUA, Y., HE, X., CHEN, Z., XIA, W., HUANG, F., AND LIU, Q. Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information. In *Proc. USENIX ATC, 2014*.

[18] GUO, F., AND EFSTATHOPOULOS, P. Building a high-performance deduplication system. In *Proc. USENIX ATC, 2011*.

[19] HUANG, P., WU, G., HE, X., AND XIAO, W. An aggressive worn-out flash block management scheme to alleviate ssd performance degradation. In *Proc. ACM EuroSys, 2014*.

[20] KACZMARCZYK, M., BARCZYNSKI, M., KILIAN, W., AND DUBNICKI, C. Reducing impact of data fragmentation caused by in-line deduplication. In *Proc. ACM SYSTOR, 2012*.

[21] LILLIBRIDGE, M., ESHGHI, K., AND BHAGWAT, D. Improving restore speed for backup systems that use inline chunk-based deduplication. In *Proc. USENIX FAST, 2013*.

[22] LILLIBRIDGE, M., ESHGHI, K., BHAGWAT, D., DEOLALIKAR, V., TREZISE, G., AND CAMBLE, P. Sparse indexing: large scale, inline deduplication using sampling and locality. In *Proc. USENIX FAST, 2009*.

[23] LIM, H., FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. SILT: A memory-efficient, high-performance key-value store. In *Proc. ACM SOSP, 2011*.

[24] MANDAGERE, N., ZHOU, P., SMITH, M. A., AND UTTAMCHANDANI, S. Demystifying data deduplication. In *Proc. ACM/IFIP/USENIX Middleware, 2008*.

[25] MEISTER, D., KAISER, J., AND BRINKMANN, A. Block locality caching for data deduplication. In *Proc. ACM SYSTOR, 2013*.

[26] MEISTER, D., KAISER, J., BRINKMANN, A., CORTES, T., KUHN, M., AND KUNKEL, J. A study on data deduplication in hpc storage systems. In *Proc. IEEE SC, 2012*.

[27] MEYER, D. T., AND BOLOSKY, W. J. A study of practical deduplication. In *Proc. USENIX FAST, 2011*.

[28] MIN, J., YOON, D., AND WON, Y. Efficient deduplication techniques for modern backup operation. *Computers, IEEE Transactions on 60*, 6 (June 2011), 824–840.

[29] NAM, Y., LU, G., PARK, N., XIAO, W., AND DU, D. H. Chunk fragmentation level: An effective indicator for read performance degradation in deduplication storage. In *Proc. IEEE HPCC, 2011*.

[30] NAM, Y. J., PARK, D., AND DU, D. H. Assuring demanded read performance of data deduplication storage with backup datasets. In *Proc. IEEE MASCOTS, 2012*.

[31] PATTERSON, D. A., GIBSON, G., AND KATZ, R. H. A case for redundant arrays of inexpensive disks (RAID). In *Proc. ACM SIGMOD, 1988*.

[32] PAULO, J. A., AND PEREIRA, J. A survey and classification of storage deduplication systems. *ACM Comput. Surv. 47*, 1 (June 2014), 11:1–11:30.

[33] QUINLAN, S., AND DORWARD, S. Venti: a new approach to archival storage. In *Proc. USENIX FAST, 2002*.

[34] STRZELCZAK, P., ADAMCZYK, E., HERMAN-IZYCKA, U., SAKOWICZ, J., SLUSARCZYK, L., WRONA, J., AND DUBNICKI, C. Concurrent deletion in a distributed content-addressable storage system with global deduplication. In *Proc. USENIX FAST, 2013*.

[35] TARASOV, V., JAIN, D., KUENNING, G., MANDAL, S., PALANISAMI, K., SHILANE, P., TREHAN, S., AND ZADOK, E. Dmdedup: Device mapper target for data deduplication. In *2014 Ottawa Linux Symposium*.

[36] WALLACE, G., DOUGLIS, F., QIAN, H., SHILANE, P., SMALDONE, S., CHAMNESS, M., AND HSU, W. Characteristics of backup workloads in production systems. In *Proc. USENIX FAST, 2012*.

[37] WILDANI, A., MILLER, E. L., AND RODEH, O. Hands: A heuristically arranged non-backup in-line deduplication system. In *Proc. IEEE ICDE, 2013*.

[38] XIA, W., JIANG, H., FENG, D., AND HUA, Y. Silo: a similarity-locality based near-exact deduplication scheme with low ram overhead and high throughput. In *Proc. USENIX ATC, 2011*.

[39] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proc. USENIX FAST, 2008*.

Chronicle: Capture and Analysis of NFS Workloads at Line Rate

Ardalan Kangarlou, Sandip Shete, and John D. Strunk
NetApp, Inc.

Abstract

Insights from workloads have been instrumental in hardware and software design, problem diagnosis, and performance optimization. The recent emergence of software-defined data centers and application-centric computing has further increased the interest in studying workloads. Despite the ever-increasing interest, the lack of general frameworks for trace capture and workload analysis at *line rate* has impeded characterizing many storage workloads and systems. This is in part due to complexities associated with engineering a solution that is tailored enough to use computational resources efficiently yet is general enough to handle different types of analyses or workloads.

This paper presents *Chronicle*, a high-throughput framework for capturing and analyzing Network File System (NFS) workloads at line rate. More specifically, we designed Chronicle to characterize NFS network traffic at rates above 10Gb/s for days to weeks. By leveraging the actor programming model and a pluggable, pipelined architecture, Chronicle facilitates a highly portable and scalable framework that imposes little burden on application programmers. In this paper, we demonstrate that Chronicle can reconstruct, process, and record storage-level semantics at the rate of 14Gb/s using general-purpose CPUs, disks, and NICs.

1 Introduction

The storage industry is in a state of flux. As new workloads emerge and the characteristics and economics of the storage media change, it is vital to reevaluate the design of storage systems. Many of yesterday's caching, prefetching, and data tiering techniques have limited applicability to today's workloads and hardware. The recent emergence of software-defined data centers and data-driven management that support a more application-centric view of storage has further increased the interest in workloads. The latest trends aside, characterizing traditional workloads and storage systems is also critical. Designing benchmarks for legacy workloads and troubleshooting deployed systems, product and service selection, capacity planning, and billing all hinge on some understanding of workloads.

Despite the ever-increasing interest, the lack of high-quality workload traces and general workload analysis frameworks have been major stumbling blocks in characterizing storage workloads and systems. Ideally, workload traces should be thorough and fine-grained enough to accurately capture the dynamics of the workloads. An I/O-by-I/O view of workloads undoubtedly provides richer insights compared to views based on aggregate statistics or sampling. Additionally, the trace collection and analysis procedure should cause the least amount of interference with the systems under study.

This paper presents *Chronicle* [2], a high-throughput framework for capturing and analyzing workloads at line rate for an extended period of time. Specifically, we designed Chronicle to characterize Network File System (NFSv3) traffic at rates above 10Gb/s for days to weeks. Chronicle runs as a Linux-based middlebox that *passively* monitors network traffic via network taps or port mirroring. The most important aspect of Chronicle is that, through deep packet inspection (DPI), it can reconstruct storage protocol semantics at line rate. Chronicle has the flexibility to capture long-term traces, perform real-time analytics on the in-flight network traffic, or do some combination of both.

We favored a middlebox approach over instrumenting NFS servers or clients because it is independent of the systems under study; and more importantly, it has no impact on system performance. We also opted for a solution based on commodity hardware. Although there are very efficient solutions based on specialized hardware, such as FPGA packet capture cards, monitoring in the networking hardware (e.g., [5, 35]), or GPUs (e.g., MIDeA [33] and PacketShader [21]), these systems tend to be limited in scope. Capture cards are generally limited to timestamping and DMAing packets; networking hardware is generally optimized to perform a few operations like lookup, filtering, and counting; and GPUs excel for applications that conform to the single instruction, multiple threads (SIMT) mode of programming.

Earlier efforts in fields such as software routing, network security, and software-defined networking (SDN) have shown the applicability of general-purpose hardware for high-speed packet processing. For instance, RouteBricks [16] achieves a throughput of 10Gb/s, 6.4Gb/s, and 1.4Gb/s for forwarding, routing, and IPsec encryption of 64B packets respectively. In this paper we demonstrate that general-purpose hardware can also handle more complex operations like TCP reassembly, pattern matching, data checksumming, compression, real-time analysis, and trace storage at rates higher than 10Gb/s.

Parallelizing packet processing using multicore architectures has been the focus of many efforts in the past [12, 16, 19, 27, 29, 34]. Partitioning and pipelining work across threads, judicious placement and scheduling of threads for better cache hit rates, and minimizing synchronization overhead (e.g., by using lock-free data structures for thread communication) are a few examples of the techniques discussed in the literature. However, constructing such carefully engineered systems imposes great burdens on programmers, because it requires intimate knowledge of hardware platforms and careful management of shared state and resources among threads. These designs often result in systems that are heavily tailored to specific hardware platforms and require manual tuning.

To address these challenges, we developed a user-space programming library, called *Libtask*, which hides such complexities from application programmers. Libtask is based on the actor model paradigm [22] and enables a lock-free, pluggable, pipelined architecture for applications that use it. There are several advantages to this architecture: (1) applications built on top of Libtask are completely decoupled from the underlying hardware, resulting in highly portable and scalable software; (2) interactions among threads are well-defined, thus reducing the possibility of concurrency bugs; and (3) supporting different types of input sources, output formats, analyses, and protocols, beyond what we demonstrate with NFS, simply involves plugging in the right module in the pipelined architecture.

To the best of our knowledge, Chronicle is the first system of its kind to show the applicability of the actor programming model to workload capture and analysis. Another novel aspect of Chronicle is that we extend zero-copy packet parsing to what is considered the application layer in the OSI reference model [36]. Therefore, there is no need to copy packet payloads to a contiguous buffer to reconstruct storage-level semantics. We demonstrate the versatility of the Chronicle framework by describing the implementation of two pipelines, one for trace capture and one for characterizing NFS workloads. These pipelines can operate at the rate of 14Gb/s using only 8 cores, a testament to the framework's efficiency. We have successfully deployed Chronicle in a number of production environments. Our intent is to create a comprehensive trace library that represents different classes of workloads across the industries that constitute our customer base.

The rest of this paper is organized as follows: Section 2 describes some of the main differences between Chronicle and earlier work. Section 3 presents a high-level overview of the Chronicle architecture. Sections 4 and 5 describe the implementation of Libtask and Chronicle pipelines respectively. Section 6 highlights unique aspects of Chronicle relative to other frameworks. Section 7 presents a comprehensive evaluation of Libtask and Chronicle, and Section 8 briefly discusses of some of the insights gained through implementing and using Chronicle.

2 Related Work

Capture and analysis of network storage workloads (e.g., NFS and CIFS) have been the focus of a few efforts in the past [8, 18, 26]. Of these efforts, Driverdump [8], a system based on modifying the network driver to directly store packets in the pcap format, is the most powerful software-only solution that can operate at the rate of 1.4Gb/s. It is unfair to directly compare the performance of Chronicle to these systems because of the hardware advances. Instead, we would like to highlight the unique features of Chronicle that have advanced the state of the art in capture and analysis of network-attached storage (NAS) workloads. These features can be summarized as (1) TCP reassembly; (2) inline parsing; and (3) efficient trace storage. As a result, Chronicle can characterize workloads at higher rates, for a longer time, and with better coverage of I/O operations compared to all the previous efforts.

The use of multiple cores for efficient packet processing is an active area of research in packet forwarding and software routing [10, 12, 16, 17, 21, 27, 32]. These efforts differ from Chronicle in that they typically do not perform any DPI and are limited to parsing the network header. This simple difference, however, has great implications for Chronicle with respect to programmability and functionality.

Kernel frameworks, such as the elegant and modular kernel-mode Click [25], require expert knowledge to extend them in a performance-optimized way. Extending such frameworks with arbitrary types of processing (e.g., custom or preexisting libraries for parsing, compression, etc.) can be especially daunting for nonexperts. The recent port of netmap [31] to user-mode Click along with techniques such as batching packet processing and recycling allocated memory, have improved the forwarding throughput by 10x, close to the throughput of Click's kernel-mode implementation [32]. In Section 6, we extensively compare the implementation of Chronicle with a few well-known packet-processing frameworks and demonstrate that our actor model framework facilitates *implicit* concurrency, serialization, and batching to achieve high throughput.

Dobrescu, Argyraki et al. [15] proposed a framework to eliminate the "tedious manual tuning" that underlay RouteBricks [16]. They devised a formula to identify the optimal parallelization strategy when packet-processing elements can be cloned or pipelined across cores. This type of framework tends to be effective in scenarios where the exact processing cost of each packet is known. In Chronicle's application scenario, per-packet processing cost can be quite variable, because factors such as packet reordering and the type of the NFS operation embedded in a packet affect the processing overhead.

Many papers and projects (e.g., [1, 13, 29, 34]) have addressed efficient use of multicore architectures for DPI or network monitoring. In addition to supporting lower rates, many of these systems have a much narrower scope than Chronicle because (1) their implementations are tied to specific multicore architectures; (2) many do not do TCP reassembly; and (3) DPI is not performed on the whole packet payload. De Sensi [14] addresses some of these limitations by leveraging structured parallel programming on top of FastFlow [7].

DPI on FastFlow is similar to Chronicle, in that both define higher-level abstractions for users to represent a workflow. The main difference lies in the programming model. For example, actors cannot share state in the actor model paradigm (Section 4). Another difference is zero-copy parsing beyond the network layer by Chronicle. Unfortunately, a direct comparison of the throughput of the two frameworks is not possible because DPI on FastFlow was evaluated using HTTP network traces, as opposed to *live* NFS traffic, and presumably with less CPU-intensive processing compared to our evaluation scenario; however, this framework could operate at 11Gb/s.

3 Chronicle Overview

This section outlines at a high level the design and architecture of Chronicle. It also describes and justifies some of the design decisions we made to address many challenges of workload characterization at line rate. We especially highlight three important challenges: (1) DPI to construct application layer semantics; (2) trace storage at line rate; and (3) efficient use of CPU cores.

Although it is not unique to Chronicle, it is important to point out that performing DPI to construct application layer semantics is more involved than simply examining a few bytes of packets beyond the network header. This complexity is due to the fact that the TCP/IP layer is completely oblivious to the nature of the application layer data it transports. For instance, to characterize NFS traffic over TCP/IP, Chronicle needs to handle situations where an RPC protocol data unit (PDU) starts in the middle of a packet or crosses multiple packets. Therefore, unlike high-speed packet forwarding and routing, this type of DPI requires reassembly of TCP segments and *stateful* parsing *across* packets. Additionally, TCP reassembly should cope with packet loss and packet retransmissions.

Another important challenge is trace storage at rates higher than 10Gb/s. At such high rates, storage bandwidth can easily become a cause for concern. We could use a high-end array of disks or SSDs, but that would conflict with our goal of using affordable off-the-shelf hardware. Additionally, workload capture for an extended period of time at these rates requires a considerable amount of storage. For example, capturing network traces using a standard tool like tcpdump at 10Gb/s for a week requires more than 750TB of storage.

We use three techniques to address these data storage challenges. The first technique is to prune the raw data that we capture off the wire. One by-product of performing DPI inline is that we can identify fields of interest in the stream of bytes we capture. For example, in the context of our NFS workload capture implementation, Chronicle records several fields of interest in the network header and almost all of the RPC and NFS fields. The second technique is inline checksumming of the NFS read and write data, which results in substantial savings over storing the raw data for data deduplication analysis [28]. The third technique is to perform inline compression prior to writing traces. By directly writing traces in the DataSeries [9] format, we can leverage DataSeries' inline compression, nonblocking I/O, and delta encoding functions, which reduce both the bandwidth and capacity requirements of trace storage. These techniques collectively reduce the amount of data recorded to a rate that a single standard disk can handle.

Although performing DPI along with inline compression and checksumming help to alleviate the storage bottleneck issues, these techniques come at the expense of increased CPU utilization. As illustrated by other high-throughput systems such as PacketShader [21] and RouteBricks [16], excessive processing at high rates can easily make CPU the bottleneck and hurt the overall throughput of the system. For instance, RouteBricks achieves a throughput of 10Gb/s for forwarding 64B packets, but performing more complex operations like routing or IPsec dropped the throughput to 6.4Gb/s and 1.4Gb/s respectively. Similarly, with netmap [31] packets can be received at 14.88Mpps (at 10Gb/s), but Open vSwitch [30] packet forwarding on top of netmap drops the throughput by more than 75% [32]. Considering the more complex nature of the operations performed by Chronicle, efficient use of CPU cores is critical.

The rest of this section describes a host of techniques that prevent CPU from becoming the bottleneck. The first technique, and arguably the most important one, is the use of the *Libtask* library. Section 4 describes the implementation of Libtask in detail. Here we briefly discuss our main objectives in designing Libtask and its place in the Chronicle architecture. Libtask's main purpose is to provide seamless scalability to many cores. It enables a pluggable, pipelined architecture, in which each module performs a different task in parallel. Applications written on top of Libtask can then use all the cores in the system without any knowledge of underlying hardware, such as the number of cores or their topology. The seamless scalability to many cores results in great portability for the Chronicle software. Additionally, the pluggable, pipelined architecture results in a very flexible framework in which, by chaining the right set of modules, Chronicle can capture traces, do statistical analysis, or perform some combination of both. Section 5.1 covers the Chronicle pipelines extensively.

The second technique is zero-copy packet parsing at both the application and network layers. As packets pass through different modules in our pipelined architecture, each module parses a specific layer (e.g., the network, RPC, or NFS layer) or performs some kind of computation based on information from previous pipeline modules (e.g., checksumming of the read/write data or compression). Therefore, to keep the overhead of the pipelined architecture low, it is imperative to avoid any sort of copying between different modules. Section 5.2 elaborates on our zero-copy packet parsing method.

The third technique is the use of custom network drivers, which allows a user-space application to bypass the kernel when reading packets. Techniques such as DPDK [3] and netmap [31] are proposed to eliminate most of the overhead associated with packet processing in standard operating systems, like the overhead of copying packets, memory allocation for packet descriptors (e.g., sk_buff structures in Linux), and interrupt processing. Our Chronicle implementation uses netmap to read packets from the NICs.

4 Libtask Library

We developed Libtask as a general actor model (AM) [22] library that facilitates seamless scalability to many cores. Central to the AM paradigm are the concepts of *actor*, *task*, and communication among actors. An actor refers to a computational agent that processes tasks. Each task is addressed to a target actor and includes some *message*, which is the information to be shared with the target [6]. As actors process the messages in tasks, the computation in an AM system advances. Processing a task by an actor can lead to sending a message (either to itself or to some other actor), creation of new actors, or actor replacement.

Two aspects of the AM programming that make it highly attractive to high-throughput computing are no sharing of state and asynchronous communication among actors. In this paradigm, the only way in which actors can affect each other is through sending messages (as opposed to sharing variables). The outcome is a very modular design in which bugs caused by concurrent execution can be easily avoided. Asynchronous communication among actors is necessary for an actor to send a message to itself and is desirable for our purposes because actors do not block to receive acknowledgements from targets. These properties enable a highly scalable and programmer-friendly framework in which many actors can be created and pipelined to carry out tasks in parallel.

Many languages such as D, Erlang, and Scala (with Akka toolkit) have borrowed concepts from the AM framework. Additionally, there are languages like Go that are based on the somewhat similar paradigm of Communicating Sequential Processes (CSP) [23]. Instead of relying on existing AM frameworks, we decided to implement our own standalone AM library in C++ for better performance and a finer level of control. Libtask is quite lightweight and small (fewer than 2,000 lines of code).

The rest of this section describes a few Libtask constructs such as Process, Scheduler, and Message. A Libtask Process is equivalent to an actor, and its implementation can be thought of as an event-driven state machine that performs a certain task. A Process has complete ownership of the data it processes. Therefore, there is no sharing of state among different Processes, as specified by the actor model. Each Process has a queue for receiving incoming Messages and is runnable as long as there is a pending Message in its queue.

A Scheduler's job is to schedule and run Processes. Each Scheduler has a queue of runnable Processes. On the occasion that the queue becomes empty, the Scheduler may steal a Process from other Schedulers. The Scheduler's run queue can become empty either as a re-

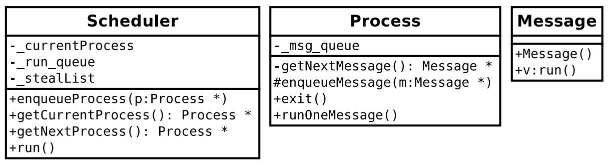

Scheduler
-_currentProcess
-_run_queue
-_stealList
+enqueueProcess(p:Process *)
+getCurrentProcess(): Process *
+getNextProcess(): Process *
+run()
Process
-_msg_queue
-getNextMessage(): Message *
#enqueueMessage(m:Message *)
+exit()
+runOneMessage()
Message
+Message()
+v:run()

Figure 1: Simplified UML diagram for Libtask.

sult of exhausting its list of runnable processes or due to Process-stealing. Upon getting scheduled, a Process runs a bounded number of Messages, where each Message is run to completion before the next is processed. The Scheduler is implemented as a POSIX thread, and there are typically as many Schedulers as there are logical cores (i.e., hardware threads) in the system.

We have developed two versions of Libtask for balancing load across cores. In one version, Schedulers are pinned to distinct logical cores and prefer to steal Processes from Schedulers running in the same NUMA node (i.e., the same CPU), thus preserving warm caches. In the other version, Schedulers are not tied to specific cores and perform NUMA-agnostic Process-stealing. Section 7.1 compares the performance of these two versions relative to implementations in Erlang and Go.

Processes use Messages as the communication mechanism between themselves. The main purpose of a Message is to specify the action to be performed by the next Process in the pipeline. A Message can contain the data to be passed or some reference to it. In either case, a Message exchange signals the transfer of data ownership between the sender and the target. Figure 1 illustrates the simplified UML diagram for the Libtask classes described earlier. One point to note is that Process::enqueueMessage is implemented as a protected method. This is to ensure that the sending and receiving Processes agree on the exchanged Message type or types. All subclasses of Process have a public wrapper method for Process::enqueueMessage (one per Message type) to force type checking at compile time.

5 Chronicle Implementation

We have implemented Chronicle as multiple pipelines of Libtask Processes. Each Process corresponds to a *module* in the Chronicle architecture, and performs functions such as parsing, computation, and trace storage. All modules and all messages exchanged between them are implemented as subclasses of Process and Message respectively. Section 5.1 describes Chronicle pipelines and Section 5.2 describes the zero-copy parsing method used by different pipeline modules.

5.1 Chronicle Modules and Pipelines

Figure 2(a) depicts the high-level view of the Chronicle architecture. This figure shows a few Packet Reader and Network Parser modules and a few Chronicle pipelines. Each pipeline itself is made up of more modules, as illustrated in Figure 2(b). The rest of this section describes the functions of each module and its role in the overall architecture. Discussion of the Messages passed between modules is postponed to Section 5.2.

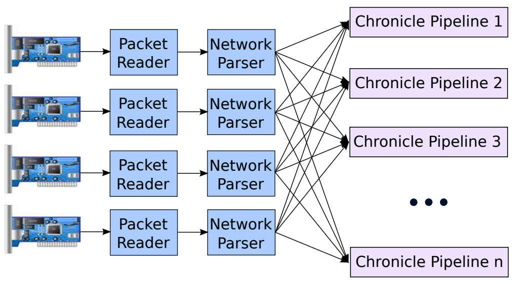
A Packet Reader (Reader) module reads Ethernet frames from a NIC using the netmap [31] drivers mentioned in Section 3. We made small changes to netmap to support jumbo frames and larger buffer sizes. Our implementation dedicates one Reader per NIC. However, for modern NICs that have multiple queues, it is possible to dedicate one Reader per queue for faster processing of the packets. Each Reader polls the corresponding NIC, timestamps all the available packets, and copies them to an internal packet buffer pool.

The main functions of a Network Parser module are parsing the network header portion of a packet and multiplexing further processing across different Chronicle pipelines. A Network Parser parses L2, L3, and L4 headers in an Ethernet frame and retrieves information such as source and destination IP addresses and port numbers, TCP sequence number, and TCP payload offset. It then uses the 5-tuple of source IP address, destination IP address, source port number, destination port number, and transport protocol to delegate further processing to one of the Chronicle pipelines. To avoid cross-pipeline communication or locking, Network Parser designates the same pipeline to process the packets belonging to either direction of a connection.

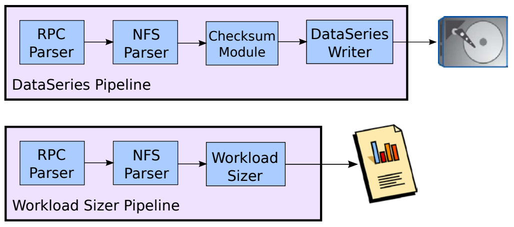
Figure 2(b) illustrates two examples of the pipelines that Chronicle currently supports. The *DataSeries Pipeline* is the pipeline of choice for trace capture at high rates due to the reasons mentioned in Section 3. We use the *Workload Sizer Pipeline* as an example of a pipeline whose purpose is to perform real-time analytics on the NFS traffic. The rest of this section describes these pipelines and their constituent modules.

5.1.1 Trace Capture Pipeline

The DataSeries pipeline receives a stream of packets on one end and generates traces in the DataSeries format [9] on the other end. The DataSeries format is characterized by efficient storage of structured serial data. Each DataSeries trace file is composed of a series of records, where each record is in turn composed of a series of fields. The records of the same type are organized into groups of extents, which are similar to tables in databases. For example, in our application scenario, we have one extent type for storing network-level infor-



(a) Chronicle modules and pipelines



(b) Different Chronicle pipelines

Figure 2: Overview of the Chronicle architecture.

mation, another extent type for storing RPC header information, and additional extent types for different NFS operation types. The records in these extents correspond to a packet, to an RPC PDU, and to an NFS operation respectively. We use a common field called *record_ID* to link related records in the network, RPC, and NFS extents. The DataSeries Writer module at the end of the DataSeries pipeline is responsible for storing traces in the DataSeries format. This module makes extensive use of a few features of DataSeries such as inline lzf compression, relative packing (delta encoding), and unique value packing.

For DataSeries Writer to store all the fields of interest in the DataSeries format, it has to rely on information provided by the preceding modules in the pipeline, starting with the RPC Parser module. This module performs the following main functions: (1) filtering of TCP and RPC traffic; (2) reassembly of TCP segments; (3) detection and parsing of RPC headers; (4) construction of RPC PDUs; and (5) matching RPC replies with the corresponding calls.

The RPC Parser module is the key module that facilitates DPI on NFS traffic. To perform DPI, this module needs to have some kind of receive-side TCP functionality to handle in-order, out-of-order, and retransmitted packets. Given that Chronicle passively captures network traffic via network taps or port mirroring, it is possible for an RPC Parser to see an acknowledgement for a TCP segment that will be seen in the future or that will never be seen. Under these circumstances, we could not rely on a standard TCP implementation and had to develop a custom TCP reassembly facility.

Packet losses and out-of-order packets directly impact the performance of the RPC Parser module and the overall throughput of Chronicle. In the absence of any losses and out-of-order packets, the identification of RPC header in the byte stream is very straightforward, because the length of a PDU is part of the header. Advancing from the current RPC header by the length of a PDU results in finding the next RPC header in the byte stream. This technique works well not only for cases where an RPC header starts immediately after the TCP header but also for commonly occurring cases where a PDU starts in the middle of a packet, when a packet contains multiple PDUs, or when a PDU spans multiple packets. When an RPC Parser uses this technique to find RPC headers, we deem that it is operating in the *fast mode*. Unfortunately, this technique falters in the event of packet loss or out-of-order delivery of packets and causes the module to enter the *slow mode*. In the slow mode, the RPC Parser module has to scan the byte stream and perform pattern matching to find an RPC header based on its signature. Once a header is found, RPC Parser can return to the fast mode if a complete PDU is present.

The NFS Parser module is responsible for parsing the NFS fields in an RPC PDU. The values for these fields are provided to the DataSeries Writer module to supply the records for the NFS operation-specific extents in a DataSeries trace file. The Checksum module operates on NFS read and write PDUs and computes 64-bit checksums for 512B read/write data blocks at 512B-aligned offsets. These checksums are also passed to the DataSeries Writer module to be stored in a data checksum extent. The checksums computed by this module can be used for online or offline data deduplication analysis [28].

In addition to supporting NICs with netmap drivers for input, Chronicle supports input from NICs or files through the standard pcap interface. It also supports writing traces in the pcap format. We will not elaborate these capabilities much further for the following reasons: (1) pcap NIC interfaces are quite lossy at high data rates; (2) the focus of this paper is to characterize *live* NFS network traffic; and (3) trace storage in the pcap format is quite bulky and requires further parsing of the data. However, these capabilities demonstrate the flexible nature of our pluggable, pipelined architecture where supporting new input sources, output formats, or protocols merely involves plugging the right set of modules in the right place in the pipeline.

5.1.2 Workload Sizer Pipeline

Another example of our flexible, pipelined architecture is a pipeline for sizing storage workloads. Workload sizing is a pre-sales practice in the storage industry to identify the right platform for a given workload. A sizer typically takes as input workload-specific information such as the rate of I/Os, the random read working set size, and the ratio of random reads and writes, and generates as output the number of heads and spindles as well as the estimated CPU and disk utilization levels for different storage platforms.

The main function of the Workload Sizer module is to generate a workload profile that will be used as input to an off-box sizer. This module processes each I/O request and leverages synopsis data structures [20] due to their speed in absorbing updates and their small memory footprint. This module also performs top-k analysis [11] and quantile calculation. Other examples of real-time analysis that Chronicle can support are pipelines to determine the data deduplication rate, the hottest files by the number of bytes or requests, and the most active clients. The insights from these pipelines are helpful in dynamically tuning a storage system.

5.2 Zero-Copy Packet Parsing

Zero-copy parsing at the network level is a standard practice and has been used extensively in operating systems and packet processing frameworks to avoid the cost of data copy between different modules. Our contribution is that we extend zero-copy parsing to the application layer. Our approach is novel in that it does not require copying packet payloads to a contiguous buffer to reconstruct application layer semantics.

The key idea behind our parsing technique is to maintain ancillary data structures on top of the packet buffer pool. Each entry in the buffer pool has a corresponding, fixed *packet descriptor* structure that serves as a handle to a particular buffer pool entry and holds all network-level information about a packet (either the data itself or its offset). Packet descriptors are allocated once at the beginning of a Chronicle run as opposed to upon every packet arrival.

Upon receipt of a packet, the Packet Reader module passes the corresponding descriptor to the Network Parser module. The Network Parser module populates most entries in the descriptor and passes it along to the appropriate Chronicle pipeline. The RPC Parser module then chains packet descriptors belonging to the same flow based on their TCP sequence number values. Prior to chaining, the RPC Parser may adjust packet descriptors to ensure that no two descriptors overlap in the TCP sequence number space.[1] When an RPC Parser finds an RPC PDU, it creates another ancillary data structure called a *PDU descriptor*. Each PDU descriptor holds RPC- and NFS-level information and points to the chain of packet descriptors that constitute the PDU. The RPC Parser then passes PDU descriptors to the next module in the pipeline. Packet or PDU descriptors passed between modules are the embodiment of the actor model messages described in Section 4.

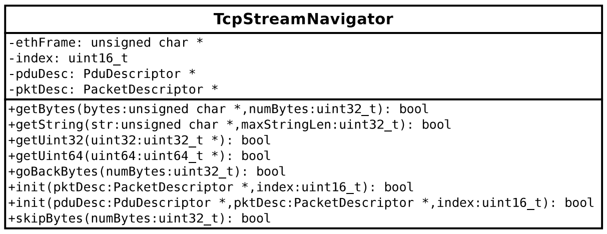

```
TcpStreamNavigator
-ethFrame: unsigned char *
-index: uint16_t
-pduDesc: PduDescriptor *
-pktDesc: PacketDescriptor *
+getBytes(bytes:unsigned char *,numBytes:uint32_t): bool
+getString(str:unsigned char *,maxStringLen:uint32_t): bool
+getUint32(uint32:uint32_t *): bool
+getUint64(uint64:uint64_t *): bool
+goBackBytes(numBytes:uint32_t): bool
+init(pktDesc:PacketDescriptor *,index:uint16_t): bool
+init(pduDesc:PduDescriptor *,pktDesc:PacketDescriptor *,index:uint16_t): bool
+skipBytes(numBytes:uint32_t): bool
```

Figure 3: Simplified UML diagram for the facility that navigates TCP byte streams.

The main enabler for the application layer zero-copy parsing is the implementation of a facility for traversing packet payloads. Figure 3 presents a simplified UML diagram for this facility. This facility maintains a point of reference, which consists of a packet descriptor and a byte offset in the payload, and uses the TCP information to retrieve certain bytes in the byte stream. We implemented an XDR parser on top of this facility for parsing the RPC header and NFS fields. One important aspect of this facility is that it enables parsing data (e.g., a single field or a group of fields as in the RPC header) that cross multiple packets. This capability is unique to Chronicle and does not exist in previous NFS tracing efforts (e.g., [8, 18]) and in standard tools like Wireshark. Another advantage is that it enables skipping all nonrelevant bytes for the DPI task at hand without any data copy.

6 Comparison with Other Frameworks

This section compares and contrasts Chronicle with the implementation of a few high-throughput packet-processing frameworks. On the surface, there are many similarities between Chronicle and frameworks like Click [25]. For instance, a Click router consists of a number of modules called elements. These elements can get pipelined, packets can get multiplexed across pipelines of elements, and there is zero-copy packet parsing across elements. Additionally, elements can run in the context of multiple scheduler threads [12]. However, there are some subtle differences, particularly with re-

[1] This condition may occur as a result of TCP retransmissions.

spect to the application scenario and programmability, which are highlighted in the rest of this section.

Latency vs. Throughput: For a software router like Click, low latency in the processing path can be as crucial as high throughput. The processing path in Click typically consists of a sequence of push and pull elements, where each element either pushes a packet to a downstream element or pulls a packet from an upstream element. *Queue* elements are typically used only when there are transitions between pull or push paths or when multiple paths converge to temporarily store packets.

Because only the source of a push path and the sink of a pull path are schedulable elements, other elements in the path must run in the context of the same thread that schedules the source or sink element [12]. This implementation minimizes thread communication, reduces scheduling overhead and cache conflicts, and imposes minimal queuing delay, which together reduce processing latency. However, to improve throughput, recent efforts [24, 32] have suggested better use of I/O and computation batching so that an element can process multiple packets at a time. In our application scenario, achieving high throughput is the primary objective and Chronicle's actor model architecture, with the implicit queues at independently schedulable processing elements, facilitates *seamless* I/O and computation batching.

Explicit vs. Implicit Parallelism: Despite some similarities, parallelism in Click and Chronicle is different in a number of ways. First, every module in Chronicle is schedulable and can run on any core. Second, with the exception of Packet Readers, when a Chronicle module gets scheduled, it is guaranteed that it has some useful work to do. This is a side effect of our implementation, where a Libtask Process gets placed on a Scheduler's queue only when it has a pending Message, and the fact that Processes only "push" Messages to each other. Third, for some frameworks, certain layouts of modules require use of thread-safe queues or modules. A positive aspect of our actor model implementation is that such complexities are not exposed to the users of Chronicle because the framework itself provides *implicit* parallelization and serialization.

Network vs. Application Layer: Differences between packet processing at the network and application layers explain some of the design decisions behind Chronicle. For instance, parsing a network header is generally not CPU-intensive enough to justify the use of multiple cores per packet. Therefore, spatial assignment techniques (e.g., NetSlices [27] and TNAPI [19]) that impose fixed mappings between packets and cores are very efficient for parsing network headers. On the other hand, these techniques may result in load imbalances and CPU underutilization when processing is expensive or variable. In fact, as we discovered during the evaluation of Chronicle (Section 7), in some scenarios, pinning threads to cores may have adverse effects on throughput. Another issue is that parsing at the application layer requires a framework to be general enough to support different application layer constructs beyond just packets (e.g., RPC PDUs). Our general actor model framework again seamlessly facilitates efficient use of cores as well arbitrary types of application layer constructs.

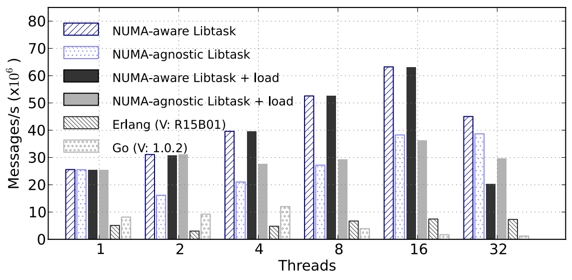
7 Evaluation

This section presents a comprehensive evaluation of Libtask and Chronicle. For these experiments, Libtask and Chronicle run on a server with two Intel Xeon E5-2690 2.90GHz CPUs. Each CPU has 8 cores (16 logical cores or hardware threads). The server is configured with 128GB of 1600MHz DDR3 DRAM memory (64GB per CPU). Additionally, it has two dual-port Intel® 82599EB 10GbE NICs, which allows capture from two tapped links or four mirrored links. The storage configuration consists of ten 3TB SATA disks. The total cost of our setup amounted to about $10,000. Section 7.2 illustrates that Chronicle can support network rates higher than 10Gb/s with a much less powerful hardware configuration. The server runs on a 3.2.32 Linux kernel with a patched ixgbe driver to support netmap. The NFS server was a NetApp® FAS6280 with two 10GbE NICs.

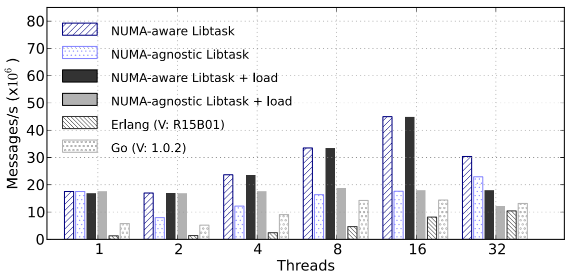
7.1 Libtask Evaluation

We used two microbenchmarks to measure the performance of Libtask against similar frameworks in Erlang (version R15B01) and Go (version 1.0.2). These evaluations also compare the performance of the NUMA-aware and NUMA-agnostic versions of Libtask. In the *Message Ring* benchmark, 1,000 Processes form a ring and pass approximately 100 million Messages around the ring, so that there are 100 outstanding Messages within the ring at any given time. In the *All-to-All* benchmark, 100 Processes send approximately 100 million Messages to each other in a random way.

Figure 4 presents the number of Messages exchanged per second for different implementations as the number of Scheduler threads varies. The results are for averages of 10 runs. For all configurations, the NUMA-aware Libtask performs the best, and both Libtask implementations outperform implementations in Erlang and Go, because Libtask is a much leaner messaging framework with none of the overhead associated with copying message data, running inside of a virtual machine, or activities like garbage collection. The drop between the 16- and 32-thread configurations for the NUMA-aware Libtask is a result of cross-socket communication. Although



(a) Message Ring benchmark



(b) All-to-All benchmark

Figure 4: Libtask evaluation.

these benchmarks do not reflect CPU-intensive tasks performed by Chronicle, they are indicative of the rate at which Libtask can distribute tasks across the cores.

To test Libtask under a more realistic setup where competing load is present, we ran one CPU-intensive thread in the background for the "+*load*" configurations of Figure 4. Because this thread is not pinned to any core, it only degrades the 32-thread setup for the NUMA-aware configurations. One interesting finding for the 32-thread setup is that NUMA-awareness degraded throughput for Message Ring. This is because for the NUMA-aware setup, at any given time one Scheduler was pinned to the same core as the competing thread. The interference resulted in a *convoy effect* for the Message Ring benchmark that hurt the overall throughput.

Another interesting finding is that for the 2-, 4-, and 8-thread NUMA-agnostic configurations, adding the extra load improved the throughput for both benchmarks. This effect is a direct consequence of the competing thread pushing a larger number of Schedulers to run on the same CPU, resulting in better cache locality for them.

The impact of the extra load suggests possible improvements to the NUMA-aware version of Libtask: (1) pinning a Scheduler to a CPU, not to a core, to alleviate the convoy effect; and (2) taking into account the communication patterns of Processes to reduce the cross-socket communication. A comprehensive analysis of these enhancements is left as future work.

7.2 Chronicle Evaluation

We chose to evaluate Chronicle using the DataSeries pipeline because it was more CPU- and disk-intensive than the analysis pipelines. Figure 7 shows the experiment setup. In this setup, a client machine was directly connected to an NFS server via two 10Gb/s links. The server running Chronicle received network traffic on both directions of the client-server links using two fiber taps. For all the experiments described in this section, we used two Chronicle pipelines (one pipeline per client-server link). Hereafter, for brevity, we use the term *core* when referring to logical cores.

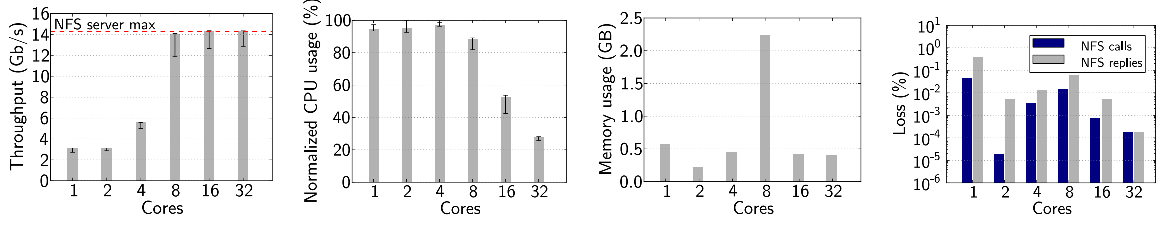
We experimented with many configurations to stress Chronicle. We observed that a mix of NFS read and write workloads resulted in the highest rates for both throughput and I/Os per second (IOPS) on the NFS server. The results presented in this section were all generated using 30-minute, constant-rate fio [4] workload runs. Interestingly, we obtained better results with NUMA-agnostic Libtask due to the convoy effect described in Section 7.1. The competing activities in the trace capture scenario were threads used by DataSeries for compression and I/O as well as applications like Apache that ran in the background. Therefore, we present results for this configuration only. This section measures the performance of Chronicle for a number of metrics, including multicore scalability, CPU and memory usage, and the success rate in capturing and parsing NFS operations.

7.2.1 Maximum Throughput

Figure 5(a) shows the maximum *sustained* throughput rates as we varied the number of cores used by Chronicle.[2] The sustained throughput rates are characterized by constant utilization of the buffer pool (Section 5.2). Therefore, Chronicle should handle these workload rates for an infinitely long duration. This also means that Chronicle can support higher data rates at the expense of higher buffer pool utilization, albeit for a bounded amount of time.

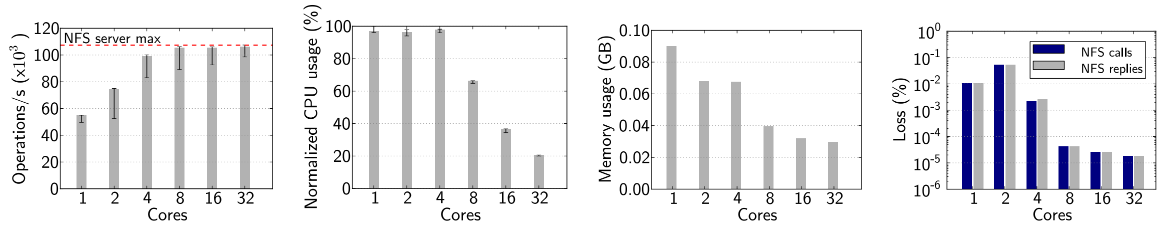
As shown in Figure 5(a), Chronicle with one core could support 3.05Gb/s. Adding a second core did not help much with throughput, although it did help with better coverage (Figure 5(d)). We suspect that polling the NICs by the four Packet Reader modules left little time for other modules. Near maximum CPU utilization for these configurations, shown in Figure 5(b), illustrates this point. However, for the 4- and 8-core configurations,

[2]The bars in figures 5(a), 5(b), 6(a), and 6(b) denote the average values, and the error bars show the minimum and maximum.



(a) Max. sustained throughput (b) CPU utilization (c) Max. buffer pool utilization (d) NFS capture loss rate

Figure 5: Maximum sustained throughput evaluation.



(a) Max. sustained IOPS (b) CPU utilization (c) Max. buffer pool utilization (d) NFS capture loss rate

Figure 6: Maximum sustained IOPS evaluation.

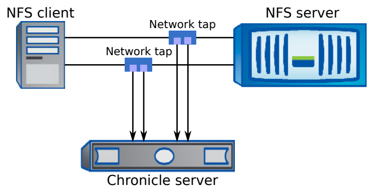


Figure 7: Experiment setup.

there were enough spare CPU cycles to sustain 5.43Gb/s and the near maximum 13.68Gb/s respectively. Adding an extra thread per core or the second CPU (i.e., the 16- and 32-core configurations) did not significantly increase the maximum sustained throughput because with 8 cores we could almost handle the maximum rate supported by the NFS server.

Figure 5(c) shows the highest usage of the buffer pool to handle the maximum throughput configurations. Although our application scenario is mostly concerned with high throughput and not processing latency (except in reading packets), the relatively low buffer utilization suggests that Chronicle processed packets very quickly. It is worth noting that the 16- and 32-core configurations had considerably less memory utilization than the 8-core configuration, because with more computational resources Chronicle could process packets much faster.

Another important metric is the loss rate in capturing NFS operations. These losses can happen either as a result of Packet Readers not getting scheduled fast enough to empty the NIC ring buffers or as a result of capture via lossy methods (e.g., port mirroring). We compared the number of NFS operations seen by the NFS server with the number of operations captured in the DataSeries traces to measure Chronicle's loss rate. For all configurations in Figure 5(d), Chronicle had a negligible loss rate. Most notably, for the 32-core configuration at 14.0Gb/s, Chronicle missed only 84 out of the total 48,600,042 NFS operations. An interesting conclusion we can draw from the results in Figure 5 is that a hardware configuration with 1GB of RAM dedicated for Chronicle, and an 8-core CPU with hyper-threading enabled, should handle 14Gb/s relatively loss-free, provided that there is a high-quality data feed (Section 7.2.3).

7.2.2 Maximum IOPS

The goal of the experiments described in this section was to stress Chronicle with an increasingly higher number of NFS operations until Chronicle reached its limit and could no longer keep up. For the experiments described in Section 7.2.1, the NFS client issued 64KB read and write operations to maximize throughput. To maximize IOPS, the client issued 1B read and write operations. Figure 6(a) shows the maximum sustained IOPS Chronicle could handle for different numbers of cores. The results suggest that with 8 cores and only 40MB of buffer space, Chronicle could handle the maximum IOPS supported by the NFS server (106 kIOPS) relatively loss free. The CPU utilization for the 8-core setup also implies that only 5 out of 8 cores were fully utilized. Therefore, Chronicle could potentially support

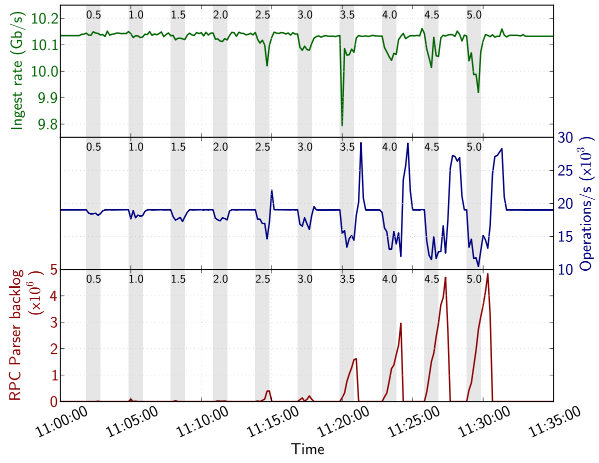


Figure 8: This figure illustrates a controlled experiment to study the impact of packet loss, when the network traffic rate is about 10Gb/s. The highlighted 1-minute intervals correspond to periods when packets got dropped at the rate of 0.5% to 5%. Although high loss rates caused significant backlog for RPC Parser, Chronicle performed well under normal network conditions and recovered quickly when the losses were intermittent.

much higher IOPS rates. In fact, when we traced a customer's metadata-intensive workload, which was generated by more than 3,000 clients, we saw that Chronicle could sustain 150 kIOPS.

The maximum sustained IOPS results illustrate an important point about Chronicle. Chronicle with 1 core can support twice as many small NFS operations as the 32-core setup of Section 7.2.1 (55,000 vs. 27,000 operations/s). Clearly, the cost of processing small PDUs is much less than that of processing large PDUs. Through CPU profiling and by examining the size of the Message queues for different modules, we have confirmed that when operating in the slow mode, RPC Parser consumes the most CPU cycles among all the modules. Recalling the discussion in Section 5.1.1, RPC Parser has to scan packet payloads in order to find the next RPC header while operating in the slow mode. When PDUs are small, it scans relatively few bytes before getting back to the fast mode. However, for large PDUs the module may potentially scan 64KB or more before it can find a header. Therefore, unlike in packet forwarding, where a high volume of small packets poses the largest overhead, a high volume of out-of-order packets belonging to large PDUs poses the biggest challenge to Chronicle.

7.2.3 The Impact of Packet Loss

The previous section discussed how packet loss can degrade Chronicle's performance. This section describes a controlled experiment to study this effect. For this experiment, we used the 32-core setup of Section 7.2.1 but made two changes. We limited the network traffic rate to about 10Gb/s, and we modified the Packet Readers to uniformly drop data packets (i.e., packets that are not empty acks) at specific time intervals. The 1-minute time intervals during which the Packet Readers induced packet loss are highlighted in Figure 8 and are annotated with the loss rates. The packet loss rates ranged from 0.5% to 5% and were interspersed with 2-minute intervals when there were no induced losses.

The top graph in Figure 8 shows the effective network traffic rate ingested by Chronicle during the course of the experiment. The middle graph illustrates the number of NFS operations that were processed by Chronicle. The dips in the graph correspond to the lower number of complete PDUs that Chronicle managed to find during the loss intervals. As loss rates increased, the dips became deeper and wider. They became deeper because there were fewer complete PDUs to be processed and they became wider because the RPC Parsers stayed in the slow mode longer (even beyond the 1-minute loss interval). However, as soon as Chronicle processed all the packets received during the loss intervals, it reverted to the fast mode and very quickly made up the lost ground. The spikes following the dips signify this behavior.

One metric that clearly captures the behavior of Chronicle under packet loss is the size of the Message backlog for the RPC Parsers (the bottom graph in Figure 8). Because an RPC Parser spends more time in the slow mode, the number of outstanding Messages in its queue grows. Although the backlog was negligible when packet loss was 2% or less, it grew very fast at higher rates. Because each Message in an RPC Parser's queue corresponds to one packet, the backlog had a direct impact on increased buffer pool utilization. The results in Figure 8 suggest that Chronicle can handle packet loss at low rates fairly well provided that the losses are intermittent and that there is a buffer pool of sufficient size to accommodate the additional processing of the out-of-order packets.

7.2.4 Trace Compression Ratio

Unsurprisingly, the size of a trace generated by Chronicle depends on the workload being captured. This section briefly discusses a 7-hour-long trace, captured from a production environment, to shed some light on the advantages of inline parsing, storing the checksums of read and write data, and inline compression over storing the raw network data, as was done in the previous efforts. For this trace, Chronicle processed 1.8TB of network traffic where 36% of the operations corresponded to NFS reads and writes. The total trace size generated

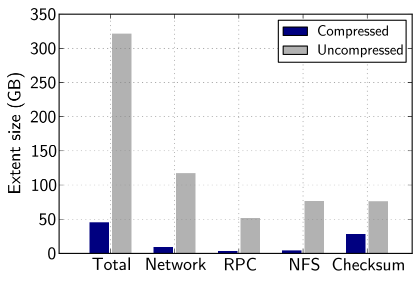


Figure 9: The size of different extents with and without compression.

by Chronicle amounted to 44.6GB, which is a 40x reduction over saving raw packets. The extents corresponding to the network header and data checksums amounted to 84% of the trace, while the extents storing the RPC and NFS fields accounted for the rest. The trace compression ratio varied from extent to extent. For instance, the extents storing the NFS, RPC, network, and checksum data had compression ratios of 20:1, 15:1, 12:1, and 3:1 respectively (Figure 9).

8 Lessons Learned

Our experience with Chronicle suggests that the actor programming model is an effective, programmer-friendly framework for workload characterization at line-rate. We believe that some of the techniques described in this paper have applicability beyond the NFS protocol. For instance, the fast- and slow-mode techniques to identify message boundaries (Section 5.1.1) have applicability to other network storage protocols such as iSCSI, SMB/CIFS, and the RESTful key-value store protocols. Similarly, the zero-copy application layer parsing technique (Section 5.2) has no limitations in supporting other protocols. The experiments described in sections 7.1 and 7.2 revealed that preserving cache locality should not come at the expense of balancing load across cores, particularly in the presence of competing load.

Chronicle has been deployed in a number of production environments to collect traces and perform sizing. For these deployments, the average traffic rates (3 to 6Gb/s) were lower than the results presented in this paper, and Chronicle could accurately capture the dynamics of the workload. One common theme among our deployments thus far has been the concentration of I/O by clients and by files. For instance, in a 3-day deployment there were 573 unique NFS clients, where the 25 most active clients accounted for more than 60% of read and write bytes served by the server. Accesses to files were also heavily concentrated. In a week-long deployment, the 25 hottest files, out of 9 million unique files accessed, accounted for 40% of total operations on the server. The insights facilitated by Chronicle can guide a storage administrator or a software-defined storage controller to dynamically tune a storage system. As an example, the knowledge of hot files and their access patterns can lead to better data caching and tiering solutions.

One powerful aspect of Chronicle is that it enables detection of problematic scenarios that are often not foreseen. For instance, we noticed that in a production environment, 8 clients out of more than 3,000 unique clients were reissuing read operations at the aggregate rate of 40 kIOPS for days. A closer examination revealed that these reads accounted for 31% of all the operations received by the server and that they were all failing due to a stale file handle!

Identifying misconfigurations is another application scenario for Chronicle. During one deployment, we observed that a server was serving *getattr* requests at the average rate of 56 kIOPS. Further analysis of the top 25 client-file pairs that were present in the getattr requests revealed that these requests were targeted at static files, with many being Linux system utilities that rarely get updated. Shockingly, there were on average 214 getattr requests per second for the top client-file pair! With insights from Chronicle traces, we were able to recommend configuring the NFS clients with correct attribute caching parameters to eliminate a sizable portion of unnecessary getattr requests. Another interesting finding was that for some clients more than 80% of read and write operations did not fall on 4096-byte boundaries. These misaligned I/Os are generally more expensive to serve by a block-based storage system and can be the result of nonbuffered I/Os at clients or incorrectly configured virtual disks for virtual machines.

9 Conclusions

This paper presented the design and implementation of Chronicle, an extremely flexible framework for characterizing workloads at line rate. We demonstrated that it is possible to capture and analyze NFS traffic at 14.0Gb/s using general-purpose CPUs, disks, and NICs. Chronicle's high-throughput architecture is facilitated by a pluggable, pipelined design that is based on actor programming model. Such a design enables seamless scalability to many cores where CPU-intensive operations such as stateful parsing, pattern matching, data checksumming, and inline compression can be done inline.

Chronicle's source code [2] is available under an academic, noncommercial license.

10 Acknowledgements

We would like to thank our shepherd, Dean Hildebrand, and anonymous reviewers for their helpful comments.

References

[1] The Bro network security monitor. https://www.bro.org.

[2] Chronicle. https://github.com/NTAP/chronicle.

[3] DPDK: Data Plane Development Kit. http://dpdk.org.

[4] fio. http://freecode.com/projects/fio.

[5] NetFlow. http://www.cisco.com/en/US/products/ps6601/products_ios_protocol_group_home.html.

[6] AGHA, G. A. Actors: A model of concurrent computation in distributed systems. Tech. Rep. 844, Massachusetts Institute of Technology, 1985.

[7] ALDINUCCI, M., DANELUTTO, M., KILPATRICK, P., AND TORQUATI, M. *FastFlow: high-level and efficient streaming on multi-core*. Parallel and Distributed Computing, Chapter 13. Wiley, 2013.

[8] ANDERSON, E. Capture, conversion, and analysis of an intense NFS workload. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies* (2009).

[9] ANDERSON, E., ARLITT, M., MORREY, C. B., AND VEITCH, A. DataSeries: an efficient, flexible data format for structured serial data. *ACM SIGOPS Operating Systems Review 43*, 1 (2009).

[10] BOLLA, R., AND BRUSCHI, R. PC-based software routers: High performance and application service support. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow* (2008).

[11] CHARIKAR, M., CHEN, K., AND FARACH-COLTON, M. Finding frequent items in data streams. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming* (2002).

[12] CHEN, B., AND MORRIS, R. Flexible control of parallelism in a multiprocessor PC router. In *Proceedings of the USENIX Annual Technical Conference* (2001).

[13] CONG, W., MORRIS, J., AND XIAOJUN, W. High performance deep packet inspection on multi-core platform. In *Proceedings of the 2nd IEEE International Conference on Broadband Network & Multimedia Technology* (2009).

[14] DESENSI, D. DPI over commodity hardware: implementation of a scalable framework using FastFlow. Master's thesis, University of Pisa and Scuola Superiore SantAnna, 2013.

[15] DOBRESCU, M., ARGYRAKI, K., IANNACCONE, G., MANESH, M., AND RATNASAMY, S. Controlling parallelism in a multicore software router. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow* (2010).

[16] DOBRESCU, M., EGI, N., ARGYRAKI, K., CHUN, B., FALL, K., IANNACCONE, G., KNIES, A., MANESH, M., AND RATNASAMY, S. RouteBricks: exploiting parallelism to scale software routers. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles* (2009).

[17] EGI, N., GREENHALGH, A., HANDLEY, M., HOERDT, M., HUICI, F., MATHY, L., AND PAPADIMITRIOU, P. Forwarding path architectures for multicore software routers. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow* (2010).

[18] ELLARD, D., LEDLIE, J., MALKANI, P., AND SELTZER, M. Passive NFS tracing of email and research workloads. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (2003).

[19] FUSCO, F., AND DERI, L. High speed network traffic analysis with commodity multi-core systems. In *Proceedings of the 10th ACM SIGCOMM Internet Measurement Conference* (2010).

[20] GIBBONS, P. B., AND MATIAS, Y. Synopsis data structures for massive data sets. In *Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms* (1999).

[21] HAN, S., JANG, K., PARK, K., AND MOON, S. PacketShader: a GPU-accelerated software router. *ACM SIGCOMM Computer Communication Review 40*, 4 (2010).

[22] HEWITT, C. Viewing control structures as patterns of passing messages. *Artificial Intelligence 20*, 3 (1976).

[23] HOARE, C. A. R. Communicating sequential processes. *Communications of the ACM 21*, 8 (1978).

[24] KIM, J., HUH, S., JANG, K., PARK, K., AND MOON, S. The power of batching in the Click modular router. In *Proceedings of the 3rd ACM Asia-Pacific Conference on Systems* (2012).

[25] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The Click modular router. *ACM Transactions on Computer Systems 18*, 3 (2000).

[26] LEUNG, A. W., PASUPATHY, S., GOODSON, G., AND MILLER, E. L. Measurement and analysis of large-scale network file system workloads. In *Proceedings of the USENIX Annual Technical Conference* (2008).

[27] MARIAN, T., LEE, K. S., AND WEATHERSPOON, H. NetSlices: scalable multi-core packet processing in user-space. In *Proceedings of the 8th ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (2012).

[28] MEYER, D. T., AND BOLOSKY, W. J. A study of practical deduplication. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies* (2011).

[29] NELMS, T., AND AHAMAD, M. Packet scheduling for deep packet inspection on multi-core architectures. In *Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (2010).

[30] PFAFF, B., PETTIT, J., KOPONEN, T., AMIDON, K., CASADO, M., AND SHENKER, S. Extending networking into the virtualization layer. In *Proceedings of ACM Workshop on Hot Topics in Networks* (2009).

[31] RIZZO, L. Netmap: a novel framework for fast packet I/O. In *Proceedings of the USENIX Annual Technical Conference* (2012).

[32] RIZZO, L., CARBONE, M., AND CATALLI, G. Transparent acceleration of software packet forwarding using netmap. In *Proceedings of the 31st IEEE International Conference on Computer Communications (IEEE INFOCOM)* (2012).

[33] VASILIADIS, G., POLYCHRONAKIS, M., AND IOANNIDIS, S. MIDeA: a multi-parallel intrusion detection architecture. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (2011).

[34] WANG, J., CHENG, H., HUA, B., AND TANG, X. Practice of parallelizing network applications on multi-core architectures. In *Proceedings of the 23rd International Conference on Supercomputing* (2009).

[35] YU, M., JOSE, L., AND MIAO, R. Software defined traffic measurement with OpenSketch. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (2013).

[36] ZIMMERMANN, H. OSI reference model - The ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications 28*, 4 (1980).

Reliable, Consistent, and Efficient Data Sync for Mobile Apps

Younghwan Go†*, Nitin Agrawal*, Akshat Aranya*, Cristian Ungureanu*
NEC Labs America*   KAIST†

Abstract

Mobile apps need to manage data, often across devices, to provide users with a variety of features such as seamless access, collaboration, and offline editing. To do so reliably, an app must anticipate and handle a host of local and network failures while preserving data consistency. For mobile environments, frugal usage of cellular bandwidth and device battery are also essential. The above requirements place an enormous burden on the app developer. We built Simba, a data-sync service that provides mobile app developers with a high-level local-programming abstraction unifying tabular and object data – a need common to mobile apps – and transparently handles data storage and sync in a reliable, consistent, and efficient manner. In this paper we present a detailed description of Simba's client software which acts as the gateway to the data sync infrastructure. Our evaluation shows Simba's effectiveness in rapid development of robust mobile apps that are consistent under all failure scenarios, unlike apps developed with Dropbox. Simba-apps are also demonstrably frugal with cellular resources.

1 Introduction

Personal smart devices have become ubiquitous and users can now enjoy a wide variety of applications, or *apps* for short, running on them. Many such apps are data-centric [2] often relying on cloud-based resources to store, share, and analyze the data. In addition to the user interface and the various features, the developer of such an app needs to build the underlying data management infrastructure. For example, in order to deliver a high-quality note-taking app such as Evernote, the developers have to build a data management platform that supports rich multimedia notes, queries on data and metadata, collaboration, and offline operations, while ensuring reliability and consistency in the face of failures. Moreover, a mobile app developer needs to meet the above requirements while also being efficient with the limited resources on mobile devices such as cellular bandwidth and battery power. The better the developer handles the above-mentioned issues the more likely the app will attract and retain users.

With the rapid growth in the number and the variety of apps in the marketplace, there is a consequent demand from practitioners for *high-level abstractions* that hide the complexity and simplify the various tasks of the app developer in managing data [6, 34, 52].

Data-sync services have emerged as an aid to developers wherein an app can offload some of its data management to a third-party service such as Dropbox, iCloud, or Google Drive. While at first such services catered to end-users who want access to their files across multiple devices, more recently such services provide SDKs for apps to use directly through CRUD (Create, Read, Update, Delete) operations [15]. Sync services are built upon decades of research on distributed and mobile data sync – from foundational work on disconnected operations [30], weakly-connected replicated storage [37, 55], and version management [42], to more recent work on wide-area database replication [58], collaborative editing [46], and caching for mobile devices [57].

The principles and mechanisms of data sync by themselves are well understood, here we do not seek to reinvent them, but a data-sync service needs to achieve a dual objective in order to be valuable to mobile apps. First, it must *transparently* handle matters of reliability, consistency, and efficiency, with little involvement from the app developer, which is challenging. As the makers of Dropbox also note, providing simplicity to users on the outside can require enormous complexity and effort *under-the-hood* [24]. Second, a data-sync service must provide a data model that is beneficial to the majority of apps; while file sync is commonplace, many apps actually operate over inter-dependent structured and unstructured data [11]. A high-level data model encompassing tables and files is of great value to app developers and the transparency must apply to this data model.

A data-sync service must preserve, on behalf of the apps, the consistency between structured and unstructured data as it is stored and shared under the presence of failures. Consider the example of photo-sharing apps such as Picasa and Instagram; typically such an app would store album information in a table and the actual images on the file system or object store. In this case, the sync service needs to ensure that there will never be dangling pointers from albums to images. Since mobile apps can crash or stall frequently for a variety of reasons [10, 50], if an app is in the middle of a data operation (a local write or sync) when a failure occurs, the sync service needs to reli-

*Work done as part of an internship at NEC Labs ably detect and recover to a consistent state. Recent work has shown that several data-sync services also spread corrupt data when used with desktop file systems [61, 62]. While services already exist for file [4, 27, 56] and table [15, 29, 43] data, none meet the above criteria.

To better understand how mobile apps and sync services maintain data consistency under failures, we conducted a study of popular mobile apps for Android including ones that use Dropbox, Parse, and Kinvey for data sync. Our study revealed that apps manage data poorly with data loss, corruption, and inconsistent behavior.

We thus built Simba to manage data for mobile apps, which provides a high-level abstraction unifying files and tables. The tables may contain columns of both primitive-type (string, integer, etc.) and arbitrary-sized objects, all accessible through a CRUD-like interface. For ease of adoption, the interface is kept similar to the ones already familiar to iOS and Android developers. Apps can construct a data model spanning both tables and objects and Simba ensures that all data is reliably, consistently, and efficiently synced with the server and other mobile devices.

Simba consists of an SDK for developing mobile apps, the Simba client app (sClient) for the mobile device, and the Simba cloud server (sCloud); all apps written with the Simba SDK, *Simba-apps*, communicate *only* with the local instance of sClient which serves as the proxy for *all* interaction with sCloud. In this paper, we focus on the transparency of the high-level abstraction as it affects Simba-apps and hence primarily discuss sClient; the entire Simba service is presented in greater detail elsewhere [45].

Through case studies we show how Simba enabled us to quickly develop several mobile apps, significantly increasing development ease and functionality. Simba-apps benefited greatly from sClient's failure transparency; an app written using Dropbox failed to preserve atomicity of an entire data object leading to torn updates and synced inconsistent data under failure. Benefiting from Simba's ability to programmatically incorporate delay-tolerant data transfer, Simba-apps also exhibited reduced network footprint and gave the device increased opportunity to turn off the cellular radio.

2 Study of Mobile App Reliability

We studied the reliability of some popular mobile apps and sync services (on Android) by systematically introducing failures – network disruption, local app crash, and device power loss – and observing the recovery outcome, if any. The apps in our study use both tables and files/objects, and rely on various existing services, *i.e.*, Dropbox, Parse, and Kinvey, for data sync. We setup two Android devices with identical apps and initial state. To simulate a network disruption we activated *airplane mode* and for crashes (1) manually *kill* the app, and (2) pull the battery out; the outcomes for the two crash tests do not differ and we thus list them once, as shown in Table 1.

For the network disruption tests, some apps (*e.g.*, Hiyu, Tumblr) resulted in loss of data if the sync failure was not handled immediately after reconnection. If the app (or the notification) was closed, no recovery happened upon restart. Some apps (UPM, TomDroid, Keepass2) did not even notify the user that sync had failed. As most apps required the user to manually resync after failure, this oversight led to data perpetually pending sync. Some apps exhibited other forms of inconsistency. For TomDroid, if the second device contacted its server for sync even in absence of changes, the delete operation blocked indefinitely. For Evernote, manual re-sync after disruption created multiple copies of the same note over and over.

For the crash tests, the table-only apps recovered correctly since they depended entirely on SQLite for crash consistency. However, apps with objects showed problematic behavior including corruption and inconsistency. For YouTube, even though the object (video) was successfully uploaded, the app lost the post itself. Instagram and Keepass2 both created a local partial object; Keepass2 additionally failed to recover the table data resulting in a dangling pointer to the object. Dropbox created a conflict file with a partial object (local corruption) and spread the corruption to the second device, just like Evernote.

Our study reveals that mobile apps still lose or corrupt data in spite of abundant prior research, analysis tools, and data-sync services. First, handling objects was particularly problematic for most apps – no app in our study was able to correctly recover from a crash during object updates. Second, instead of ensuring correct recovery, some apps take the easier route of disabling object updates altogether. Third, in several cases, apps fail to notify the user of an error causing further corruption. The study further motivates us to take a holistic approach for transparently handling failures inside a data-sync service and provide a useful high-level abstraction to apps.

3 App Development with Simba

3.1 Data Model and API

Data Model: Simba's data model is designed such that apps can store *all* of their data in a single, *unified*, store without worrying about how it is stored and synced. The high-level abstraction that enables apps to have a data model spanning tables and objects is called a Simba Table (sTable in short). To support this unified view of data management, Simba, under the hood, ensures that apps always see a consistent view of data stored locally, on the cloud, and other mobile devices.

The unit of client-server consistency in Simba is an *individual* row of an sTable (sRow in short) which consists of tabular data and all objects referred in it; objects are not shared across sRows. Simba provides causal consistency

	App	DM	Disruption	Recover Outcome	Crash	Recover Outcome
With Table Only	Fetchnotes *Kinvey (notes)*	T	Upd/Del note	✓ Auto resync	Upd/Del note	✓ All or nothing
	Syncboxapp *Dropbox (notes)*	T	Upd/Del note	✓ Auto resync	Upd/Del note	✓ All or nothing
	Township *Parse (social game)*	T	Background autosave	✗ App closed with data loss	Background autosave	✓ All or nothing
	UPM *Dropbox (pwd manager)*	T	Set pwd	✗ No notification	Set pwd	✓ All or nothing
	TomDroid *(notes)*	T	Upd/Del note	✗ No notification. Del blocked if other device syncs even w/o change	Upd/Del note	✓ All or nothing
	Hiyu *Kinvey (grocery list)*	T	Upd/Del item	✗ Change loss if app is closed during disruption	Upd/Del item	✓ All or nothing
Without Obj Update	Pinterest *(social n/w)*	T+O	Create pin-board	✓ Manual resync	Set/Del pin	✓ All or nothing
	Twitter *(social n/w)*	T+O	Post (re)tweet	✓ Manual resync	Post/Del tweet	✓ All or nothing
	Facebook *(social n/w)*	T+O	Post status Post comment	✓ Auto resync ✗ Comment loss if status is closed during disruption	Post/Del status	✓ All or nothing
	Tumblr *(blogging)*	T+O	Post image	✗ Post loss if notification or app closed during disruption	Post/Del image	✓ All or nothing
	YouTube *(video stream)*	T+O	Post/Del video	✓ Auto resync	Del video Post video	✓ All or nothing ✗ Post loss even for video-upload success
	Instagram *(social n/w)*	T+O	Post image Post comment	✓ Manual resync ✗ Comment loss if image is closed during disruption	Del image Post image	✓ All or nothing ✗ Partial image created locally in gallery = corruption
With Obj Update	Keepass2 *Dropbox (pwd mgr)*	O	Set/Del pwd	✗ No notification	Set pwd	✗ Password loss; partial object created locally in "kdbx" filesystem = corruption
	Dropbox *(cloud store)*	T+O	Upd/Del file	✓ Auto resync	Upd file	✗ Partial conflict file created; file corruption and spread to second client
	Evernote *(notes)*	T+O	Upd/Del note	✗ Manual sync creates multiple copies of same note	Upd note image	✗ Note image corrupted and spread to second client

Table 1: **Study of App Failure Recovery.** DM denotes the data model (T: tables only; O: objects only; T+O: app stores both tables and objects). ✗ denotes problem behavior and ✓ indicates correct handling. "Disruption" and "crash" columns list the workload for that test.

semantics with all-or-nothing atomicity over an sRow for *both* local and sync operations; this is a stronger guarantee than provided by existing sync services. An app can, of course, have a tabular-only or object-only schema, which Simba trivially supports. Since an sRow represents a higher-level, semantically meaningful, unit of app data, ensuring its consistency under all scenarios is quite valuable to the developer and frees her from writing complicated transaction management and recovery code. Figure 1 shows Simba's data model.

Simba currently does not provide atomic sync across sRows or sTables. While some apps may benefit from atomic multi-row sync, our initial experience has shown that ACID semantics under sync for whole tables would needlessly complicate Simba design, lead to higher performance overheads, and be overkill for most apps.

API: sClient's API, described in Table 2, is similar to the popular CRUD interface but with four additional features: 1) CRUD operations on tables *and* objects 2) operations to register tables for sync 3) upcalls for new data and conflicts 4) built-in conflict detection and support for resolution. Objects are written to, or read from, using a stream abstraction which allows Simba to support large objects; it also enables locally reading or writing only part of a large object – a property that is unavailable for BLOBs (binary large objects) in relational databases [38].

Since different apps can have different sync requirements, Simba supports per-table sync policies controlled by the app developer using the sync methods (*registerWriteSync* etc). Each sTable can specify a non-zero *period* which determines the frequency of change collection for sync. A *delay tolerance* (DT) value can be specified which gives an additional opportunity for data to be coalesced across apps before sending over the network; DT can be set to zero for latency-sensitive data. Even when apps have non-aligned periods, DT enables cross-app traffic to be aligned for better utilization of the cellular radio. If an app needs to sync data on-demand, it can use the *writeSyncNow()* and *readSyncNow()* methods. Simba's delay-tolerant transfer mechanism directly benefits from prior work [22, 49]. Since sync happens in the background, when new data is available or conflicts occur due to sync, apps are informed using *upcalls*. An app can begin and end a *conflict-resolution transaction* at-will and iterate over conflicted rows to resolve with either the local copy, the server copy, or an entirely new choice.

CRUD (on tables and objects)
createTable(TBL, schema, properties)
updateTable(TBL, properties)
dropTable(TBL)
outputStream[] ← writeData(TBL, TBLData, objColNames)
outputStream[] ← updateData(TBL, TBLData, objNames, selection)
inputStream[] ← rowCursor ← readData(TBL, projection, selection)
deleteData(TBL, selection)
Table and Object Synchronization
registerWriteSync(TBL, period, DT, syncpref)
unregisterWriteSync(TBL)
writeSyncNow(TBL)
registerReadSync(TBL, period, DT, syncpref)
unregisterReadSync(TBL)
readSyncNow(TBL)
Upcalls
newDataAvailable(TBL, numRows)
dataConflict(TBL, numConflictRows)
Conflict Resolution
beginCR(TBL)
getConflictedRows(TBL)
resolveConflict(TBL, row, choice)
endCR(TBL)

Table 2: **Simba Client Interface.** Operations available to mobile apps for managing table and object data. TBL refers to table name.

3.2 Writing a Simba App

Simba's unified API simplifies data management for apps; this is perhaps best shown with an example. We consider a photo-sharing app which stores and periodically syncs the images, along with their name, date, and location. First, create an sTable by specifying its schema:

```
sclient.createTable("album", "name VARCHAR, date
  INTEGER, location FLOAT, photo OBJECT", FULL_SYNC);
```

Next, register for read (download) and write (upload) sync. Here, the app syncs photos every 10 mins (600s) with a DT of 1 min (60s) for both reads and writes, selecting WiFi for write and allowing 3G for read sync.

```
sclient.registerWriteSync("album",600,60,WIFI);
sclient.registerReadSync("album",600,60,3G);
```

A photo can be added to the table with *writeData()* followed by writing to the output stream.

```
// byte[] photoBuffer has camera image
List<SCSOutputStream> objs = sclient.writeData("album"
     , new String[]{"name=Kopa","date=15611511","
     location=24.342"}, new String[] {"photo"});
objs[0].write(photoBuffer); objs[0].close();
```

Finally, a photo can be retrieved using a query:

```
SCSCursor cursor = sclient.readData("album", new
     String[] { "location", "photo" }, "name=?", new
     String[] { "Kopa" }, null);
// Iterate over cursor to get photo data
SCSInputStream mis = cursor.getInputStream().get(1);
```

4 Simba Design

4.1 Simba Server (sCloud)

The server is a scalable cloud store that manages data across multiple apps, tables, and clients [45]. It provides a network protocol for data sync, based on a model in which it is the responsibility of an sClient to pull updates from the server and push any local modifications, on behalf of all device-local Simba-apps; the sClient may register with the server to be notified of changes to subscribed tables.

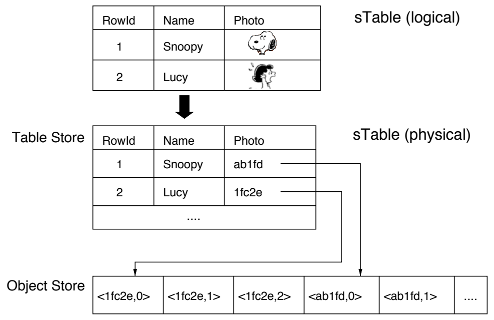


Figure 1: **Simba Client Data Store.** Table Store is implemented using a SQL database and Object Store with a key-value store based on LSM tree. Objects are split into fixed-size chunks.

Sync Protocol: To discuss sClient's design we need to refer to the semantics offered by the server through the network protocol. The server is expected to provide durability, atomicity of row updates, and multi-version concurrency control. Thus, the sClient is exposed to versions, which accompany any data in messages exchanged with the server. Simba implements a variant of *version vectors* that provides concurrency control with *causal consistency* semantics [33]. Since all sClients sync to a central sCloud, we simplify the versioning scheme to have one version number per row instead of a vector [42]. Each sRow has a unique identifier ID_{row} generated from a primary key, if one exists, or randomly, and a version V_{row}.

Row versions are incremented at the server with each update of the row; the largest row version in a table is maintained as the table version, V_{table}, allowing us to quickly identify which rows need to be synchronized. A similar scheme is used in gossip protocols [60]. Since Simba supports variable-sized, potentially large, objects, the protocol messages explicitly identify objects' partially-changed sets that need to be applied atomically.

4.2 Simba Client (sClient)

sClient allows networked Simba-apps to continue to have a local I/O model which is shown to be much easier to program for [14]; sClient insulates the apps from server and network disruptions and allows for a better overall user experience. Figure 2 shows the simplified architecture of the sClient; it is designed to run as a device-wide service which (1) provides all Simba-apps with access to their table and object data (2) manages a device-local replica to enable disconnected operations (3) ensures fault-tolerance, data consistency, and row-level atomicity (4) carries out all sync-related operations over the network. Simba-apps link with sClient through a lightweight

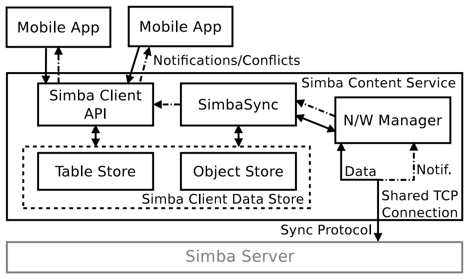


Figure 2: **Simba Client Architecture.**

library (sClientLib) which provides the Simba Client Interface (Table 2) and forwards client operations to sClient; the apps are alerted through upcalls for events (*e.g.*, new data, conflict) that happen in the background. Finally, sClient monitors liveness of apps, so that memory resources can be freed in case of app crash.

The sClient data store (§4.2.1) provides a unified abstraction over a table store and an object store. SimbaSync performs sync processing (§4.2.2) with the sCloud; for upstream sync, it collects the locally-modified data, and for downstream sync, it applies changes obtained from the server into the local store, detects conflicts, and generates upcalls to apps. The sync protocol and the local data store together provide transparent failure handling for all Simba-apps (§5). The Network Manager handles all network connectivity and server notifications for the sClient (§6); it provides an efficient utilization of the device's cellular radio through coalescing and delay-tolerance.

Implementation: sClient is currently implemented on Android, however, the design principles can be applied to other mobile platforms such as iOS. sClient is implemented as a daemon called the Simba Content Service (SCS) which is accessed by mobile apps via local RPC; on Android we use an AIDL [1] interface to communicate between the apps and the service. An alternate approach – to link directly with the app – is followed by Dropbox [16] and Parse [43] but our approach allows sClient to shape network traffic for all Simba-apps on the same device thereby benefiting from several cross-app optimizations. While the benefits of using persistent connections have been long known [35], individual apps use TCP connections in a sub-optimal manner with frequent connection establishment and teardown. sClient's design allows it to use a single persistent TCP connection to the sCloud on behalf of multiple apps; the same connection is also reused by the server for delivering notifications, providing additional savings, similar to Thialfi [9].

A misbehaving app can potentially adversely affect other Simba-apps. In practice, we believe that developers already have an incentive to write well-behaved apps to keep users satisfied. In the future, fine-grained accounting of data, similar to Android's accounting, can be built into Simba to further discourage such behavior.

4.2.1 Simba Client Data Store

The sClient Data Store (SDS) is responsible for storing app data on the mobile device's persistent storage (typically the internal flash memory or the external SD card). For Simba-apps, this means having the capability to store both tabular data and objects in a logically *unified* manner. The primary design goal for SDS is to enable, and efficiently support, CRUD operations on sRows; this requires the store to support atomic updates over the local data. Additionally, since objects are variable-sized and potentially large, the store also needs to support atomic sync of such objects. Since the store persistently stores all local modifications, a frequent query that it must efficiently support is *change detection* for upstream sync; SDS should be able to quickly determine sub-object changes. Figure 1 shows the SDS data layout.

Objects are subdivided into fixed-size chunks and stored in a key–value store (KVS) that supports range queries. The choice of the KVS is influenced by the need for good throughput for both appends and overwrites since optimizing for random writes is important for mobile apps [28]. Each chunk is stored as a KV–pair, with the key being a $\langle object\_id,\ chunk\_number \rangle$ tuple. An object's data is accessed by looking up the first chunk of the object and iterating the KVS in key order.

Local State: sClient maintains additional local state, persistent and volatile, for sync and failure handling. Two persistent per-row flags, $Flag_{TD}$ (table dirty) and $Flag_{OD}$ (object dirty), are used to identify locally-modified data, needed for upstream sync. To protect against partial object sync, we maintain for each row $Count_{OO}$, the number of objects opened for update. A write transaction for a row is considered closed when all its open objects are closed. Each row has two more persistent flags, $Flag_{SP}$ (sync pending) and $Flag_{CF}$ (conflict), which track its current sync state. Finally, an in-memory *dirty chunk table* (DCT) tracks chunks that have been locally modified but not yet synced. This obviates the need to query the store for these changes during normal operation.

Implementation: We leverage SQLite to implement the tabular storage with an additional data type representing an object identifier ($object\_id$). Object storage is implemented using LevelDB [32] which is a KVS based on a log-structured merge (LSM) tree [40]; LevelDB meets the throughput criteria for local appends and updates. LevelDB also has snapshot capability which we leverage for atomic sync. There is no native port of LevelDB for Android so we ported the original C++ LevelDB code using Android's Native Development Kit (NDK). We use one instance of LevelDB to keep objects for all tables to ensure sequential writes for better local performance [28]. Since the local state is stored in an sRow's tabular part, SQLite ensures its consistent update.

4.2.2 Sync processing

An sClient independently performs upstream and downstream sync. The upstream sync is initiated based on the specified periodicity of individual tables, and using local state maintained in ($Flag_{TD}$, $Flag_{OD}$) to determine dirty row data; these flags are reset upon data collection. For rows with dirty objects, chunks are read one-by-one and directly packed into network messages.

Since collecting dirty data and syncing it to the server may take a long time, we used the following techniques to allow concurrent operations by the foreground apps. First, sClient collects object modifications from LevelDB snapshots of the current version. As sClient syncs a modified object only after it is closed and the local state is updated (decrement $Count_{OO}$ by 1), sClient always ensures a consistent view of sRows at snapshots. Second, we allow sClient to continue making modifications while previous sync operations are in-flight; this is particularly beneficial if the client disconnects and sync is *pending* for an extended duration. These changes set sRow's local flags, $Flag_{TD}$ or $Flag_{OD}$, for collection during the subsequent sync. For this, sClient maintains a sync pending flag $Flag_{SP}$ which is set for the dirty rows, once their changes are collected, and reset once the server indicates success. If another sync operation starts before the previous one completes, rows with $Flag_{SP}$ already set are ignored.

Downstream sync is also initiated by an sClient in response to a server notification of changes to a table. The client pulls all rows that have a version greater than the local V_{table}, staging the downstream data until all chunks of a row are received and then applying it row-by-row onto the sClient data store in increasing V_{row} order.

Conflicts on upstream sync are determined through V_{row} mismatch on the server, while for downstream by inspecting the local dirty flag of received rows. To enable apps to automatically resolve [31] or present to its users, the server-returned conflicted data is staged locally by sClient and the relevant Simba-app is notified. sClient is designed to handle conflicting updates gracefully. Conflicted rows are marked ($Flag_{CF}$) to prevent further upstream sync until the conflict is resolved. However, apps can resolve conflicts at their own convenience and can continue reading and writing to their local version of the row without sync. We believe this greatly improves the user experience since apps do not have to abruptly interrupt operations when conflicts arise.

5 Transparent Failure Handling

Mobile apps operate under congested cellular networks [13], network disruptions [20], frequent service and app crashes [10], and loss of battery [44]. Mobile OS memory management can also aggressively kill apps [12].

Failure transparency is a key design objective for sClient which it achieves through three inter-related aspects. First, the mechanism is *comprehensive*: the system detects each possible type of failure and the recovery leaves the system in a well-defined state for each of them. Second, recovery leaves the system not merely in a known state, but one that obeys *high-level consistency* in accordance with the unified data model. Third, sClient is *judicious* in trading-off availability and recovery cost (which itself can be prohibitive in a mobile environment). Barring a few optimizations (discussed in §5.2), an sClient maintains adequate local metadata to avoid distributing state with the server for the purposes of recovery [41]. sClients are stateful for a reason: it allows the sync service, having many mobile clients, which can suffer from frequent failures, and a centralized server, to decouple their failure recovery thereby improving availability.

5.1 Comprehensive & High-level Consistent

sClient aims to be comprehensive in failure handling and to do so makes the use of a state machine [53]. Each successful operation transitions sClient from one well-defined state to another; failures of different kinds lead to different *faulty* states each with well-defined recovery.

We first discuss network failures which affect only the sync operations. As discussed previously, the server response to upstream sync can indicate either success or conflict and to downstream sync can indicate either success or incompletion. Table 3(a) describes sClient's status in terms of the local sync-pending state ($Flag_{SP}$) and the relevant server response (RC_O, RC_T, RU_O, RU_T); note that only a subset of responses may be relevant for any given state. Each unique state following a network disconnection, for upstream or downstream sync, represents either a no-fault or a fault situation; for the latter, a recovery policy and action is specified sClient. Tables 3 (b) and (c) specify the recovery actions taken for failures during upstream and downstream sync respectively. The specific action is determined based on a combination of the dirty status of the local data and the server response.

Crashes affect both sync and local operations and the state of the SDS is the same whether sClient, Simba-app, or the device crash. sClient detects Simba-app crashes through a signal on a *listener* and de-allocates in-memory resources for the app. Table 4 shows the recovery actions taken upon sClient restart after a crash; for a Simba-app crash, recovery happens upon its restart.

sClient handles both network failures and crashes while maintaining all-or-nothing update semantics for sRows – in all cases, the state machine specifies a recovery action that preserves the atomicity of the tabular and object data – thereby ensuring the consistency of an app's high-level unified view; this is an important value proposition of sClient's failure transparency to mobile apps. As seen in Table 4, when an object is both dirty *and* open ($Flag_{OD}$ = 1 & $Count_{OO} > 0$), a crash can lead to row inconsis-

Type	State Upon Network Disconnection	Implication	Recovery Policy	Action
Up stream	SP=0	No sync	Not needed	None (no-fault)
	SP=1, before `SyncUpResult`	Missed response	Reset & retry	SP←0, TD←1, if ∃DCT OD←1
	SP=1, after `SyncUpResult`(RC_O=0)	Completed	Roll forward	None (no-fault)
	SP=1, after `SyncUpResult`(RC_O=1)	Partial response	Reset & retry	See Table 3(b)
Down stream	Before `Notify`	No sync	Not needed	None (no-fault)
	After `Notify`	Sync needed	Normal operation	Send `SyncDownstream`
	After `SyncDownstream`	Missed response	Retry	Resend `SyncDownstream`
	After `SyncDownResult`(RU_O=0)	Completed	Roll forward	See Table 3(c)
	After `SyncDownResult`(RU_O=1)	Partial response	Reset & retry	See Table 3(c)

(a) Sync Failure Detection and Recovery Policy

Flags		Resp.	Recovery Action
TD	OD	RC_T	
0	0	*	Delete entry, SP←0
0	1	*	Delete entry, SP←0, TD←1, if ∃DCT OD←1
1	0	*	Delete entry, SP←0, TD←1
1	1	*	Delete entry, SP←0, TD←1, if ∃DCT OD←1

(b) Recovery action for `SyncUpstream`

Flags		Response		Recovery Action
TD	OD	RU_T	RU_O	
*	*	*	1	Delete entry, resend w/ new V_{table}: `SyncDownstream`
0	0	1	0	Update table data
1	*	1	0	Conflict on table data

(c) Recovery action for `SyncDownstream`

Table 3: **Network Disconnection: State Space for Failure Detection and Recovery.** CF= 0 for all the above states since sync is in-progress; OO is irrelevant. ∃DCT → Obj exists in DIRTYCHUNKTABLE. Delete entry → Delete row in TBLCONFLICT and corresponding object in LevelDB. TD: Table Dirty, OD: Object Dirty, SP: Sync Pending, RC_O: Response conflict for object, RC_T: Response conflict for table, RU_O: Response update for object, RU_T: Response update for table. Note TD and OD can be re-set to 1 after SP=1 since Simba allows local ops to safely proceed even when prior sync is in-progress. * indicates recovery action is independent.

tency, *i.e.*, a *torn* write. Similarly, a network disruption during an object sync can cause a *partial* sync; sClient detects and initiates appropriate *torn* recovery.

5.2 Judicious

sClient balances competing demands: on the one hand, normal operation should be efficient; on the other, failure recovery should be transparent and cheap. sClient maintains persistent state to locally detect and recover from most failures; for torn rows, after local detection, it recovers efficiently through server assistance. There are two kinds of tradeoffs it must make to keep recovery costs low.

5.2.1 Tradeoff: Local State vs. Network I/O

When sClient recovers from a crash, it can identify whether the object was dirty using $Flag_{OD}$ but it cannot determine whether it was completely or partially written to persistent storage; the latter would require recovery. $Count_{OO}$ counter enables making this determination: if it is set to zero, sClient can be sure that local data is consistent and avoid torn recovery using the server. The cost to sClient is an extra state of 4 bytes per row. However, one problem still remains: to sync this object, sClient still needs to identify the dirty chunks. The in-memory DCT will be lost post-crash and force sClient to either fetch all chunks from the server or send all chunks to the server for chunk-by-chunk comparison. sClient thus pays the small cost of persisting DCT, prior to initiating sync, to prevent re-syncing entire, potentially large, objects. Once persisted, DCT is used to sync dirty chunks after a crash and removed post-recovery. If sClient crashes before DCT is written to disk, it sends all chunks for dirty objects.

TD	OD	OO	SP	CF	Recovery action after crash (Flags)
0	0	=0	0	0	Do nothing
				1	Conflict upcall
0	0	=0	1	–	Restart `SyncUpstream` with table data and object if ∃DCT (TD←1, OD←1 if ∃DCT, SP←0)
0	0	>0	0	0	Do nothing (OO←0)
				1	Conflict upcall (OO←0)
0	0	>0	1	–	Restart `SyncUpstream` with table data and object if ∃DCT (TD←1, OD←1 if ∃DCT, OO←0, SP←0)
0	1	=0	0	0	Start `SyncUpstream` with full object
				1	Conflict upcall
0	1	=0	1	–	Restart `SyncUpstream` with full row (TD←1, SP←0) ← No information on which object is dirty
0	1	>0	0	*	Recover *Torn write* (OD←0, OO←0)
0	1	>0	1	–	Recover *Torn write* (OD←0, OO←0, SP←0)
1	0	=0	0	0	Start `SyncUpstream` with table data
				1	Conflict upcall
1	0	=0	1	–	Restart `SyncUpstream` with table data and object if ∃DCT (OD←1 if ∃DCT, SP←0)
1	0	>0	0	0	Start `SyncUpstream` with table data (OO←0)
				1	Conflict upcall (OO←0)
1	0	>0	1	–	Restart `SyncUpstream` with table data and object if ∃DCT (OD←1 if ∃DCT, OO←0, SP←0)
1	1	=0	0	0	Start `SyncUpstream` with full row
				1	Conflict upcall
1	1	=0	1	–	Restart `SyncUpstream` with full row (SP←0)
1	1	>0	0	*	Recover *Torn write* (TD←0, OD←0, OO←0)
1	1	>0	1	–	Recover *Torn write* (TD←0, OD←0, OO←0, SP←0)

Table 4: **Client Crash: State Space for Failure Detection & Recovery.** TD: Table Dirty, OD: Object Dirty, OO: Object Open Count, SP: Sync Pending, CF: Row Conflict; * indicates recovery action independent of flag; – indicates state with flag=1 is not possible

5.2.2 Tradeoff: Local I/O vs. Network I/O

If an object does have a non-zero $Count_{OO}$ post-crash, it is indeed torn. The most obvious way to recover torn rows is to never update data in-place in the SDS, but instead always write out-of-place first; once the data is successfully written, it can be copied to the final location similar to a write-ahead-log or journaling. Instead of paying the overhead during common-case operation, in this case, sClient takes assistance from Simba.

At any point in time, Simba has *some* consistent view of

Operation	Method	Throughput (MB/s)
Update	In-place	2.29 ± 0.08
	Out-of-place	1.37 ± 0.04
Read	In-place	3.94 ± 0.04
	Out-of-place	3.97 ± 0.07

Table 5: **Server-assisted Recovery.** Comparison of in-place and out-of-place local throughput with 1KB rows

that row; the client relies on this observation to either rollback or roll-forward to a consistent state. If sClient detect a local torn row during recovery, it obtains a consistent version of the row from the server; this is akin to rollback for aborted database transactions [36]. If the server has since made progress – the client in essence rolls forward. If the client is disconnected, recovery cannot proceed, but also does not prevent normal operation – only the torn rows are made unavailable for local updates. For comparison, we also implement an out-of-place SDS; as shown in Table 5, sClient is able to achieve 69% higher throughput with in-place updates as opposed to out-of-place updates for updating rows with 1KB objects.

6 Transparent Network Efficiency

Simba sync is designed to make judicious use of cellular bandwidth and device battery through a custom-built network protocol with two optimizations:

Delay tolerance and coalescing: typically, many apps run in the background as *services*, for example to send/receive email, update weather, synchronize RSS feeds and news, and update social networking. sClient is designed as a device-wide service so that sync data for multiple independent apps can be managed together and transferred through a shared persistent TCP connection. Further, Simba supports delay-tolerant data scheduling which can be controlled on a per-table basis. Delay tolerance and coalescing has two benefits. 1) Improved network footprint: allows data transfer to be clustered, reducing network activity and improving the odds of the device turning off the radio [49]. Control messages from the server are subject to the same measures. 2) Improved scope for data compression: outgoing data for multiple apps is coalesced to improve the compression [23].

Fine-grained change detection: an entire object need not be synced if only a part changes. Even though data is versioned per row, sClient keeps internal soft-state (DCT) to detect object changes at a configurable chunk level; Simba server does the same for downstream sync.

Implementation: Even though sRows are the logical sync unit, sClient's Network Manager packs network messages with data from multiple rows, across multiple tables and apps, to reduce network footprint. Simba's network protocol is implemented using Protobufs [7], which efficiently encodes structured data, and TLS for secure network communication; the current prototype uses two-way SSL authentication with client and server certificates.

7 Evaluation

We wish to answer the following two questions:

- Does Simba provide failure transparency to apps?
- Does Simba perform well for sync and local I/O?

We implemented sClient for Android interchangeably using Samsung Galaxy Nexus phones and an Asus Nexus 7 tablet all running Android 4.2. WiFi tests were on a WPA-secured WiFi network while cellular tests were run on 4G LTE: KT and LGU+ in South Korea and AT&T in US. Our prototype sCloud is setup using 8 virtual machines partitioned evenly across 2 Intel Xeon servers each with a dual 8-core 2.2 GHz CPU, 64GB DRAM, and eight 7200 RPM 2TB disks. Each VM was configured with 8GB DRAM, one data disk, and 4 CPU cores.

7.1 Building a Fault-tolerant App

The primary objective of Simba is to provide a high-level abstraction for building fault-tolerant apps. Evaluating success, while crucial, is highly subjective and hard to quantify; we attempt to provide an assessment through three qualitative means: (1) comparing the development effort in writing equivalent apps using Simba and Dropbox. (2) development effort in writing a number of Simba-apps from scratch. (3) observing failure recovery upon systematic fault-injection in sClient.

7.1.1 Writing Apps: Simba vs. Dropbox

Objective: is to implement a photo-sync app that stores album metadata and images. App_S is to be written using Simba and App_D using Dropbox. We choose Dropbox since it has the most feature-rich and complete API of existing systems and is also highly popular [56]; Dropbox provides APIs for files (`Filestore`) and tables (`Datastore`). App_S and App_D must provide the same semantics to the end-user: a consistent view of photo albums and reliability under common failures; we compare the effort in developing the two equivalent apps.

Summary: achieving consistency and reliability was straightforward for App_S taking about 5 hours to write and test by 1 developer. However, in spite of considerable effort (3 – 4 days), App_D did not meet all its objectives; here we list a summary of the limitations:

1. Dropbox does not provide any mechanism to consistently inter-operate the table and object stores.
2. Dropbox `Datastore` in-fact does not even provide row-level atomicity during sync (only column-level)!
3. Dropbox does not have a mechanism to handle torn rows and may sync inconsistent data.
4. Dropbox carries out conflict resolution in the background and prevents user intervention.

Methodology: we describe in brief our efforts to overcome the limitations and make App_D equivalent to App_S; testing was done on 2 Android smartphones – one as

Apps	Description	Total LOC	Simba LOC
Simba-Notes	"Rich" note-taking with embedded images and media; relies on Simba for conflict detection and resolution, sharing, collaboration, and offline support. Similar to Evernote [3]	4,178	367
Surveil	Surveillance app capturing images and metadata (*e.g.*, time, location) at frequent intervals; data periodically synced to cloud for analysis. Similar to iCamSpy [26]	258	58
HbeatMonitor	Continuously monitors and records a person's heart rate, cadence and altitude using a Zephyr heartbeat sensor [63]; data periodically synced to cloud for analysis. Similar to Sportstracklive [8]	2,472	384
CarSensor	Periodically records car engine's RPM, speed, engine load, etc using a Soliport OBD2 sensor attached to the car and then syncs to the cloud; similar to Torque car monitor [59]	3,063	384
SimbaBench	Configurable benchmark app with tables and objects to run test workloads	207	48
App_S	Simba-based photo-sync app with write/update/read/delete operations on tabular and object data	527	170
App_D	Dropbox-based photo-sync app written to provide similar consistency and reliability as App_S	602	–
sClient	Simba client app which runs as a background daemon on Android	11,326	–
sClientLib	Implements the Simba SDK for writing mobile apps; gets packaged with a Simba-app's `.apk` file	1,008	–

Table 6: **Lines of Code for Simba and Apps.** Total LOC counted using CLOC; Simba LOC counted manually

writer and the other as the reader. We were successful with **1**, **2** but not with **3**, **4**.

✓1. Consistency across stores: we store App_D images in `Filestore` and album info in `Datastore`; to account for dependencies, we create an extra `Datastore` column to store image identifiers. To detect file modifications, we maintain Dropbox *listeners* in App_D.

Writes: when a new image is added on the writer, the app on the reader receives separate updates for tables and files, Since Dropbox does not provide row-atomicity, it is possible for Simba metadata columns to sync before app data. To handle out-of-order arrival of images or album info prior to Simba metadata, we set flags to indicate tabular and object sync completion; when Simba metadata arrives, we check this flag to determine if the entire row is available. The reader then displays the image.

Updates: are more challenging. Since the reader does not know the updated columns, and whether any objects are updated, additional steps need to be taken to determine the end of sync. We create a separate metadata column (MC) to track changes to `Datastore`; MC stores a list of updated app-columns at the writer. We also issue sync of MC before other columns so that the reader is made aware of the synced columns. Since Dropbox does not provide atomicity over row-sync, the reader checks MC for every table and object column update.

Deletes: once the writer deletes the tabular and object columns, both listeners on the reader eventually get notified, after which the data is deleted locally.

✓2. Row-atomicity for tables+files: for every column update, `Datastore` creates a separate sync message and sends the entire row; it is therefore not possible to distinguish updated columns and their row version at sync. Atomic sync with Dropbox thus requires even more metadata to track changes; we create a separate table for each column as a workaround. For example, for an app table having one table and one object column, two extra tables need to be created in addition to MC.

For an update, the writer lists the to-be-synced tabular and object columns (*e.g.*, $\langle col1, col3, obj2\rangle$) in MC and issues the sync. The reader receives notifications for each update and waits until all columns in MC are received. In case a column update is received before MC, we log the event and revisit upon receiving MC. Handling of new writes and deletes are similar and omitted for brevity.

✗3. Consistency under failures: Providing consistency under failures is especially thorny in the case of App_D. To prevent torn rows from getting synced, App_D requires a separate persistent flag to detect row-inconsistency after a crash, along with all of the recovery mechanism to correctly handle the crash as described in §5. Since App_D also does not know the specific object in the row that needs to be restored, it would require a persistent data structure to identify torn objects.

✗4. Consistent conflict detection: Dropbox provides transparent conflict resolution for data; thus, detecting higher-level conflicts arising in the app's data model is left to the app. Since there is no mechanism to check for potential conflicts *before* updating an object, we needed to create a persistent *dirty* flag for each object in App_D. Moreover, an app's local data can be rendered unrecoverable if the conflict resolution occurs in the background with an "always theirs" policy. To recover from inconsistencies, App_D needs to log data out-of-place, requiring separate local persistent stores.

To meet **3.** and **4.** implied re-implementing the majority of sClient functionality in App_D and was not attempted.

7.1.2 Other Simba Apps

We wrote a number of Simba-apps based on existing mobile apps and found the process to be easy; the apps were robust to failures and maintained consistency when tested. Writing the apps on average took 4 to 8 hours depending on the GUI since Simba handled data management. Table 6 provides a brief description of the apps along with their total and Simba-related lines of code (LOC).

7.1.3 Reliability Testing

We injected three kinds of failures, network disruption, Simba-app crash, and sClient crash, while issuing local, sync, and conflict handling operations. Table 7 shows,

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x
Tab					T			T	S	S, F, R					S	S, R	S, R, F			F, R			R	R, F
Obj						C	C, O, D	C, O, D			S, D	S, D	S, D, R	S, D, T, F	S, D	S, D, R	S, D, R, F				R	R, D, F	R	R, D, F

(a) Detection. T: $Flag_{TD}$, O: $Flag_{OD}$, C: $Count_{OO}$, D: *DCT*, S: $Flag_{SP}$, R: *Server Response Table*, F: $Flag_{CF}$

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x
Tab					N			N	R	R, P					R	R, LR	R, LR, P			R, P			LR	LR, P
Obj						R	LR, SR	LR, SR			R	R	R, LR	R, LR, P	R	R, LR	R, LR, P				R	R, LR, P	LR	LR, P

(b) Recovery. N: *No Op*, R: *Reset*, P: *Propagate Conflict*, LR: *Local Recovery*, SR: *Server-assisted Recovery*

Table 7: **sClient Detection and Recovery.** The table shows detection and recovery policies of sClient for failure at read, write, syncup and syncdown operations. The operations are **a:** *read tab* **b:** *get obj readstream* **c:** *read obj* **d:** *read tab+obj* **e:** *write tab* **f:** *get obj writestream* **g:** *write obj* **h:** *write tab+obj* **i:** *syncup tab only* **j:** *syncupresult for tab only* **k:** *syncup obj only* **l:** *send objfrag for obj only syncup* **m:** *syncupresult for obj only* **n:** *get objfrag for obj only syncupresult* **o:** *syncup tab+obj* **p:** *syncupresult for tab+obj* **q:** *get objfrag for tab+obj syncupresult* **r:** *notify* **s:** *syncdown* **t:** *syncdownresult for tab only* **u:** *syncdownresult for obj only* **v:** *get objfrag for obj only syncdownresult* **w:** *syncdownresult for tab+obj syncdownresult* **x:** *get objfrag for tab+obj syncdownresult*.

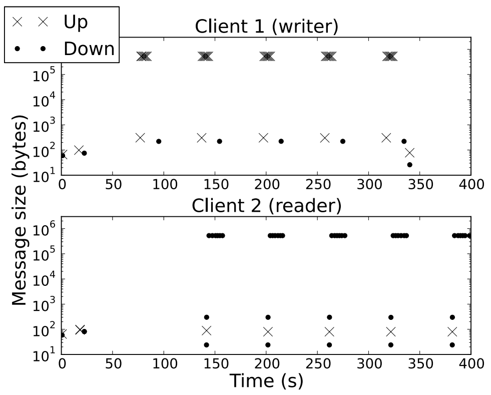


Figure 3: **Sync Network Messages.** Data and control transfer profile

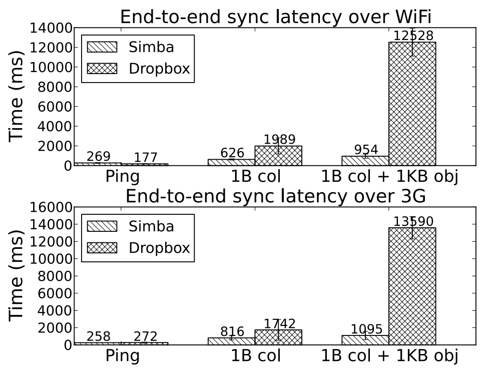


Figure 4: **Sync Network Latency.** Measured end-to-end for 2 clients

in brief, the techniques employed by sClient. For a given workload (a – x), gray cells represent unaffected or invalid scenarios, for example, read operations. A non-empty cell in detection implies that all cases were accounted for, and a corresponding non-empty cell in recovery implies corrective action was taken. The absence of empty cells indicates that sClient correctly detected and recovered from all of the common failures we tested for.

Detection: each cell in Table 7(a) lists the flags used to detect the status of tables and objects after a failure. sClient maintained adequate local state, and responses from the server, to correctly detect all failures. Change in tabular data was detected by $Flag_{TD}$ (T) for write and $Flag_{SP}$ (S) for sync as $Flag_{TD}$ is toggled at start of sync. sClient then performed a check on the server's response data (R). Sync conflict was identified by checking $Flag_{CF}$ (F). Similarly, usage of writestream and object update were detected by $Count_{OO}$ (C) and $Flag_{OD}$ (O) with the addition of DCT (D) for sync.

Recovery: each cell in Table 7(b) lists the recovery action taken by sClient from among no-op, reset, propagate, and local or server-assisted recovery. No-op (N) implies that no recovery was needed as the data was already in a consistent state. When a conflict was detected, but with consistent data, sClient propagated (P) an alert to the user seeking resolution. With the help of local state, in most cases sClient recovered locally (LR); for a torn row, sClient relied on server-assisted recovery (SR). In some cases, sClient needed to reset flags (R) to mark the successful completion of recovery or a no-fault condition.

7.2 Performance and Efficiency

7.2.1 Sync Performance

We want to verify if Simba achieves its objective of periodic sync. Figure 3 shows the client-server interaction for two mobile clients both running the SimbaBench (Table 6); on Client 1 it creates a new row with 100 bytes of table data and a (50% compressible) 1MB object every 10 seconds. Client 1 also registers for a 60-second periodic upstream sync. Client 2 read-subscribes the same table also with a 60-second period. As can be seen from the figure, the network interaction for both upstream and downstream sync shows short periodic burst of activity followed by longer periods of inactivity. Client 2's read subscription timer just misses the first upstream sync ($77s - 95s$), so the first downstream sync happens about a minute later ($141s - 157s$); for the rest of the experiment, downstream messages immediately follow the upstream ones confirming that Simba meets this objective.

We want to evaluate Simba's sync performance and

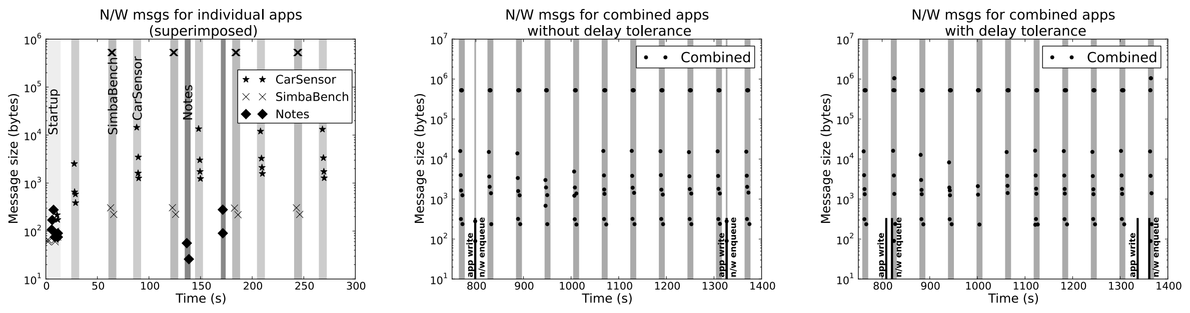


Figure 5: **Network Transfer For Multiple Apps.**

how it compares with Dropbox. Figure 4 compares the end-to-end sync latency of Simba and Dropbox over both WiFi and 4G; y-axis is time taken with standard deviation of 5 trials. For these tests we run two scenarios, both with a single row being synced between two clients: 1) with only a 1-byte column, and 2) with one 1-byte column and one 1KB object. The two clients were both in South Korea. The Dropbox server was located in California (verified by its IP address) whereas the Simba server was located on US east coast. As a baseline, we also measured the ping latency from clients to servers. Figure 4 shows that the network latency ("Ping") is a small component of the total sync latency. For both the tests, Simba performs significantly better than Dropbox; in case 1), by about 100% to 200%, and in case 2) by more than 1200%. Since Dropbox is proprietary we not claim to fully understand how it functions; it very well might be overloaded or throttling traffic. The experiment demonstrates that Simba performs well even when giving control of sync to apps.

We want to test how quickly Simba resolves conflicts for a table with multiple writers. Figure 6 shows this behavior. The x-axis shows the number of clients (min. 2 clients needed for conflict) and the y-axis shows the average time to converge (sec) and standard deviation over 5 trials. For "theirs", the server's copy is chosen *every time* and hence no changes need to be propagated back; for "mine", the local copy is chosen every time and re-synced back to the server. The "no conflicts" case is shown to establish a baseline – a normal sync still requires changes to be synced to the server; "mine" always and "theirs" always represent the worst-case and the best-case scenarios respectively with typical usage falling somewhere in between. The figure shows that for a reasonable number (*i.e.*, 5) of collaborating clients, as the number of conflict resolution rounds increases, it does not impose a significant overhead compared to baseline sync, even when selecting the server's copy; when selecting the local copy, conflict resolution is fairly quick.

7.2.2 Network Efficiency

We want to evaluate Simba's impact on network efficiency. Three apps were chosen for this experiment that generate data periodically: CarSensor app in replay mode generating about 250 byte rows every second, SimbaBench set to create 1MB rows (50% compressible) every 10s, and an app that simulates the behavior of Simba-Notes, by generating ~300 byte of data using Poisson distribution with a mean value of 300s and using a fixed seed for random number generation. CarSensor and SimbaBench run with a periodic upstream sync of 60s.

Figure 5 shows a scatter plot of the data transfer profile of the apps; y-axis is message size on a log scale, and x-axis is time in seconds. The colored bands are meant to depict temporal clusters of activity. The "Startup" band shows the one-time Simba authentication and setup, and sync registration messages for the tables. We ran the Simba apps (a) individually, (b) concurrently with Simba-Notes's DT=0, and (c) concurrently with Simba-Notes's DT=60s. Figure 5(a) shows the super-imposition of the data transfer profile when the apps were run individually, to simulate the behavior of the apps running without coordination. As also seen in the figure, while it is possible for uncoordinated timers to coincide, it is unlikely; especially so when the period is large compared to the data transfer time. Aperiodic apps like Simba-Notes also cause uncoordinated transfers. Uncoordinated transfers imply frequent radio activity and energy consumed due to large tail times. In Figure 5(b), all apps are run concurrently. The events generated by Simba-Notes are annotated. We see that the network transfers of CarSensor and SimbaBench are synchronized, but Simba-Notes still causes network transfer at irregular times (the thin bands represent network transfers by Simba-Notes). In Figure 5(c), we run an experiment similar to (b) but this time Simba-Notes employs a delay tolerance of 60s; its network activity is delayed until the next 60s periodic timer along with all pending sync activity (notice the absent thin bands). The resulting data transfer is clustered, increasing the odds of the radio being turned off. The x-axes in (b) and (c) start around 800s as we measured after a few minutes of app start.

7.2.3 Local I/O Performance

Our objective is to determine whether sClient's local performance is acceptable for continuous operation, especially since storage can be a major contributor to performance of mobile apps [28]. SimbaBench issues writes,

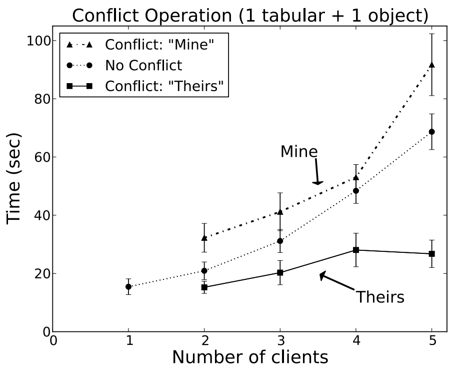


Figure 6: **Time Taken for Conflict Convergence.**

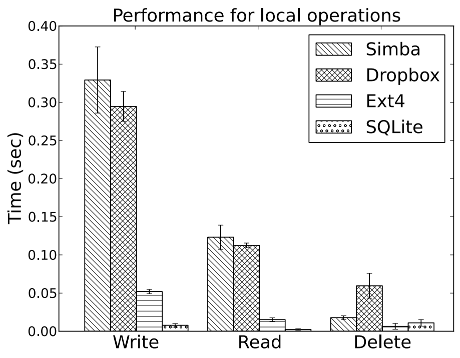


Figure 7: **sClient Local I/O Performance.**

reads, and deletes for one row of data containing one 1MB object for both sClient with Dropbox (Core API). Figure 7 shows average times and standard deviation over 5 trials; sClient is about 10% slower than Dropbox for both writes and reads, primarily due to IPC overhead as sClient is a background service on Android while Dropbox directly accesses the file system. sClient performs better for deletes through lazy deletion – data is only marked as deleted but physically removed only after sync completion. sClient and Dropbox both perform several additional operations over Ext4 and SQLite; we provide this comparison only as a baseline.

8 Related Work

Data sync and services: sync has been much studied in the context of portable devices including seminal work on disconnected operations [30], weakly-consistent replicated storage [37, 55], and data staging [18, 57].

In terms of failure transparency, Bayou [55] provides a limited discussion of its crash recovery through a write log but it does not handle objects. LBFS [37] atomically commits files on writeback, preventing corruption on crash or disruption, but does not handle tables. We find that for most apps, handling the dependencies – between tabular and object data – is the biggest source of inconsistency.

Of the existing services, Dropbox is the most comprehensive but still does not support sync atomicity for objects and tables, breaking failure transparency for several fault conditions. iCloud also provides separate mechanisms for a key-value interface and file sync. Mobius [14] provides a CRUD API to a table-sync store but does not support objects at all. Similar to Simba, Parse [43] and Kinvey [29] are mobile backend-as-a-service offering GUI integration, administration, and limited data management; they only support tables and provide last-writer-wins semantics which is inadequate for many apps. No sync service provides delay-tolerant transfer.

Fault tolerance: ViewBox [62] integrates a desktop FS with a data-sync service so as to sync only consistent views of the local data; the paper also shows how Dropbox spreads local file corruption which ViewBox addresses through checksums. Simba focuses on providing transparent fault-handling to apps; while ViewBox works only for files, Simba spans both files and tables.

Storage unification: prior work for desktop file systems has considered database integration but without network sync or a unified API. InversionFS [39] uses Postgres to implement a file system with transactional guarantees and fine-grained versioning. TableFS [51] uses separate storage pools for metadata (an LSM tree) and files to improve its own performance through metadata operations. KVFS [54] stores file data and file-system metadata both in a single key–value store built on top of VT-Trees, a variant of LSM trees, which enable efficient storage for objects of various sizes; VT-Trees can be used to build a better-performing sClient data store, in the future.

Mobile data transfer: Recent research has characterized and optimized data transfer for mobile environments [21, 25, 47], especially the adverse effects of small, sporadic transfers [17, 48]; SPDY [5] extends HTTP for better compression and multiplexes requests over a single connection to save round trips. This large body of networking research has inspired Simba's network protocol.

9 Conclusions

Building high-quality data-centric mobile apps invariably mandates the developer to build a reliable and efficient data management infrastructure – a task for which few are well-suited. Mobile app developers should not need to worry about the complexities of network and data management but instead be able to focus on what they do best – implement the user interface and features – and deliver great apps to users. We built Simba to empower developers to rapidly develop and deploy robust and efficient mobile apps; through its mobile client daemon, sClient, it provides background data sync with flexible policies that suit a large class of mobile apps while transparently handling failures and efficiently utilizing mobile resources. We plan to release Simba's source code; please check with the contact author (Nitin Agrawal) for further details.

10 Acknowledgements

We thank our FAST reviewers and shepherd, Jason Nieh, for their valuable feedback. We thank Dorian Perkins for his work on Simba Cloud and the IST group at NEC Labs for its setup; Simba Cloud was also evaluated using NMC PRObE [19]. Younghwan thanks the ICT R&D program of MSIP/IITP, Republic of Korea (14-911-05-001).

References

[1] Android Developers Website. http://developer.android.com/index.html.

[2] Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2011 – 2016. http://tinyurl.com/cisco-vni-12.

[3] Evernote App. http://evernote.com.

[4] Google Drive. https://developers.google.com/drive/.

[5] Google SPDY. https://developers.google.com/speed/spdy.

[6] Onavo. www.onavo.com.

[7] Protocol Buffers. http://code.google.com/p/protobuf.

[8] SportsTrackLive Mobile App. http://www.sportstracklive.com/help/android.

[9] A. Adya, G. Cooper, D. Myers, and M. Piatek. Thialfi: a client notification service for internet-scale applications. In *SOSP '11*.

[10] S. Agarwal, R. Mahajan, A. Zheng, and V. Bahl. Diagnosing mobile applications in the wild. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 22. ACM, 2010.

[11] N. Agrawal, A. Aranya, and C. Ungureanu. Mobile data sync in a blink. In *HotStorage '13*, San Jose, California.

[12] Android Developers. Processes and Threads. http://developer.android.com/guide/components/processes-and-threads.html.

[13] A. Balasubramanian, R. Mahajan, and A. Venkataramani. Augmenting mobile 3g using wifi. In *MobiSys '10*, pages 209–222, 2010.

[14] B.-G. Chun, C. Curino, R. Sears, A. Shraer, S. Madden, and R. Ramakrishnan. Mobius: unified messaging and data serving for mobile apps. In *MobiSys '12*, 2012.

[15] Dropbox. Dropbox Datastore API. dropbox.com/developers/datastore, July 2013.

[16] Dropbox Sync API. dropbox.com/developers/sync.

[17] H. Falaki, D. Lymberopoulos, R. Mahajan, S. Kandula, and D. Estrin. A first look at traffic on smartphones. In *IMC '10*, 2010.

[18] J. Flinn, S. Sinnamohideen, N. Tolia, and M. Satyanarayanan. Data Staging on Untrusted Surrogates. In *FAST '03*, San Francisco, CA, Apr. 2003.

[19] G. Gibson, G. Grider, A. Jacobson, and W. Lloyd. Probe: A thousand-node experimental cluster for computer systems research. volume 38, June 2013.

[20] Y. Go, Y. Moon, G. Nam, and K. Park. A disruption-tolerant transmission protocol for practical mobile data offloading. In *MobiOpp'12*, 2012.

[21] K. Ha, P. Pillai, W. Richter, Y. Abe, and M. Satyanarayanan. Just-in-time provisioning for cyber foraging. In *MobiSys '13*, pages 153–166, New York, NY, USA, 2013. ACM.

[22] S. Hao, N. Agrawal, A. Aranya, and C. Ungureanu. Building a Delay-Tolerant Cloud for Mobile Data. In *IEEE MDM*, June 2013.

[23] D. Harnik, R. Kat, O. Margalit, D. Sotnikov, and A. Traeger. To Zip or not to Zip: Effective Resource Usage for Real-Time Compression. In *FAST 2013*, Feb 2013.

[24] D. Houston. Dropbox dbx: Developer conference keynote. http://vimeo.com/70089044, 2014.

[25] J. Huang, F. Qian, Y. Guo, Y. Zhou, Q. Xu, Z. M. Mao, S. Sen, and O. Spatscheck. An in-depth study of lte: Effect of network protocol and application behavior on performance. In *SIGCOMM '13*, 2013.

[26] iCamSpy App. Audio Video Surveillance CCTV. http://www.icamspy.com/.

[27] iCloud for Developers. developer.apple.com/icloud.

[28] H. Kim, N. Agrawal, and C. Ungureanu. Revisiting Storage for Smartphones. In *FAST '12*, February 2012.

[29] Kinvey. http://kinvey.com.

[30] J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Trans. Comput. Syst.*, 10(1), February 1992.

[31] P. Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. In *USENIX ATC '95*, Berkeley, CA, USA, 1995.

[32] LevelDB: A Fast and Lightweight Key/Value Database Library. code.google.com/p/leveldb.

[33] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. A short primer on causal consistency. *USENIX ;login magazine*, 38(4), Aug. 2013.

[34] Z. Miners. Dropbox adds new tools to make syncing smarter. http://www.pcworld.com/article/2043980/dropbox-adds-new-tools-to-make-syncing-smarter.html, July 2013.

[35] J. C. Mogul. The case for persistent-connection http. In *SIGCOMM*, pages 299–313, 1995.

[36] 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, March 1992.

[37] A. Muthitacharoen, B. Chen, and D. Mazières. A Low-Bandwidth Network File System. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, pages 174–187, Lake Louise, Alberta, Oct. 2001.

[38] MySQL. MySQL BLOB and TEXT types. `http://dev.mysql.com/doc/refman/5.0/en/string-type-overview.html`.

[39] M. A. Olson. The Design and Implementation of the Inversion File System. In *USENIX Winter '93*, San Diego, CA, Jan. 1993.

[40] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree. In *Acta Informatica*, June 1996.

[41] J. K. Ousterhout. The role of distributed state. In *In CMU Computer Science: a 25th Anniversary Commemorative*, page pp. ACM Press, 1991.

[42] D. S. Parker Jr, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *Software Engineering, IEEE Transactions on*, (3):240–247, 1983.

[43] Parse. `parse.com`.

[44] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In *EuroSys '12*, pages 29–42, New York, NY, USA, 2012. ACM.

[45] D. Perkins, N. Agrawal, A. Aranya, C. Yu, Y. Go, H. Madhyastha, and C. Ungureanu. Simba: Tunable End-to-End Data Consistency for Mobile Apps. In *Proceedings of the European Conference on Computer Systems (EuroSys '15)*, Bordeaux, France, April 2015.

[46] K. P. Puttaswamy, C. C. Marshall, V. Ramasubramanian, P. Stuedi, D. B. Terry, and T. Wobber. Docx2go: collaborative editing of fidelity reduced documents on mobile devices. In *MobiSys '10*, 2010.

[47] F. Qian, K. S. Quah, J. Huang, J. Erman, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck. Web caching on smartphones: ideal vs. reality. In *MobiSys '12*, 2012.

[48] F. Qian, Z. Wang, Y. Gao, J. Huang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck. Periodic transfers in mobile applications: network-wide origin, impact, and optimization. In *WWW '12*, 2012.

[49] M.-R. Ra, J. Paek, A. B. Sharma, R. Govindan, M. H. Krieger, and M. J. Neely. Energy-delay tradeoffs in smartphone applications. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 255–270, New York, NY, USA, 2010. ACM.

[50] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan. Automatic and scalable fault detection for mobile applications. In *ACM MobiSys*, 2014.

[51] K. Ren and G. Gibson. Tablefs: Enhancing metadata efficiency in the local file system. In *USENIX ATC*, June 2013.

[52] D. ROWINSKI. Why the facebook-parse deal makes parse's rivals very, very happy. `http://readwrite.com/2013/04/29/parse-acquisition-makes-its-rivals-very-happy`, April 2013.

[53] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, Dec. 1990.

[54] P. J. Shetty, R. P. Spillane, R. R. Malpani, B. Andrews, , J. Seyster, and E. Zadok. Building Workload-Independent Storage with VT-Trees. In *FAST '13*, February 2013.

[55] 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 *SOSP '95*, 1995.

[56] The Cristian Science Monitor. Dropbox has hit the 175-million-user mark, cofounder says. `http://tinyurl.com/mlz8x3c`, July 2013.

[57] N. Tolia, J. Harkes, M. Kozuch, and M. Satyanarayanan. Integrating Portable and Distributed Storage. In *FAST '04*, pages 227–238, San Francisco, CA, April 2004.

[58] N. Tolia, M. Satyanarayanan, and A. Wolbach. Improving mobile database access over wide-area networks without degrading consistency. In *Proceedings of the 5th international conference on Mobile systems, applications and services*, pages 71–84. ACM, 2007.

[59] Torque App. Engine Performance and Diagnostic Tool. `http://torque-bhp.com/`.

[60] R. van Renesse, D. Dumitriu, V. Gough, and C. Thomas. Efficient Reconciliation and Flow Control for Anti-entropy Protocols. In *LADIS '08*, 2008.

[61] Yupu Zhang, Chris Dragga, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. *-Box: Towards Reliability and Consistency in Dropbox-like File Synchronization Services. In *Proceedings of the 5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '13)*, San Jose, California, June 2013.

[62] Yupu Zhang, Chris Dragga, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. ViewBox: Integrating Local File Systems with Cloud Storage Services. In *Proceedings of the 12th Conference on File and Storage Technologies (FAST '14)*, Santa Clara, California, February 2014.

[63] Zephyr. Zephyr HxM BT Heartrate Monitor. `http://tinyurl.com/zephyr-sensor`.

RIPQ: Advanced Photo Caching on Flash for Facebook

Linpeng Tang*, Qi Huang†★, Wyatt Lloyd‡★, Sanjeev Kumar★, Kai Li*

*Princeton University, †Cornell University, ‡ University of Southern California, ★Facebook Inc.

Abstract

Facebook uses flash devices extensively in its photo-caching stack. The key design challenge for an efficient photo cache on flash at Facebook is its workload: many small random writes are generated by inserting cache-missed content, or updating cache-hit content for advanced caching algorithms. The Flash Translation Layer on flash devices performs poorly with such a workload, lowering throughput and decreasing device lifespan. Existing coping strategies under-utilize the space on flash devices, sacrificing cache capacity, or are limited to simple caching algorithms like FIFO, sacrificing hit ratios.

We overcome these limitations with the novel Restricted Insertion Priority Queue (RIPQ) framework that supports advanced caching algorithms with large cache sizes, high throughput, and long device lifespan. RIPQ aggregates small random writes, co-locates similarly prioritized content, and lazily moves updated content to further reduce device overhead. We show that two families of advanced caching algorithms, Segmented-LRU and Greedy-Dual-Size-Frequency, can be easily implemented with RIPQ. Our evaluation on Facebook's photo trace shows that these algorithms running on RIPQ increase hit ratios up to ~20% over the current FIFO system, incur low overhead, and achieve high throughput.

1 Introduction

Facebook has a deep and distributed photo-caching stack to decrease photo delivery latency and backend load. This stack uses flash for its capacity advantage over DRAM and higher I/O performance than magnetic disks.

A recent study [20] shows that Facebook's photo caching hit ratios could be significantly improved with more advanced caching algorithms, i.e., the Segmented-LRU family of algorithms. However, naive implementations of these algorithms perform poorly on flash. For example, Quadruple-Segmented-LRU, which achieved ~70% hit ratio, generates a large number of small random writes for inserting missed content (~30% misses) and updating hit content (~70% hits). Such a random write heavy workload would cause frequent garbage collections at the Flash Translation Layer (FTL) inside modern NAND flash devices—especially when the write size is small—resulting in high write amplification, decreased throughput, and shortened device lifespan [36].

Existing approaches to mitigate this problem often reserve a significant portion of device space for the FTL (over-provisioning), hence reducing garbage collection frequency. However, over-provisioning also decreases available cache capacity. As a result, Facebook previously only used a FIFO caching policy that sacrifices the algorithmic advantages to maximize caching capacity and avoid small random writes.

Our goal is to design a flash cache that supports advanced caching algorithms for high hit ratios, uses most of the caching capacity of flash, and does not cause small random writes. To achieve this, we design and implement the novel Restricted Insertion Priority Queue (RIPQ) framework that efficiently approximates a priority queue on flash. RIPQ presents programmers with the interface of a priority queue, which our experience and prior work show to be a convenient abstraction for implementing advanced caching algorithms [10, 45].

The key challenge and novelty of RIPQ is how to translate and approximate updates to the (exact) priority queue into a flash-friendly workload. RIPQ aggregates small random writes in memory, and only issues aligned large writes through a restricted number of insertion points on flash to prevent FTL garbage collection and excessive memory buffering. Objects in cache with similar priorities are co-located among these insertion points. This largely preserves the fidelity of advanced caching algorithms on top of RIPQ. RIPQ also lazily moves content with an updated priority only when it is about to be evicted, further reducing overhead without harming the fidelity. As a result, RIPQ approximates the priority queue abstraction with high fidelity, and only performs consolidated large aligned writes on flash with low write amplification.

We also present the Single Insertion Priority Queue (SIPQ) framework that approximates a priority queue with a single insertion point. SIPQ is designed for memory-constrained environments and enables the use of simple algorithms like LRU, but is not suited to support more advanced algorithms.

RIPQ and SIPQ have applicability beyond Facebook's photo caches. They should enable the use of advanced caching algorithms for static-content caching—i.e., read-only caching—on flash in general, such as in Netflix's flash-based video caches [38].

We evaluate RIPQ and SIPQ by implementing two families of advanced caching algorithms, Segmented-LRU (SLRU) [26] and Greedy-Dual-Size-Frequency (GDSF) [12], with them and testing their performance on traces obtained from two layers of Facebook's photo-

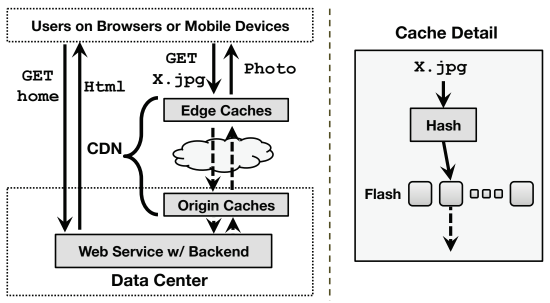


Figure 1: Facebook photo-serving stack. Requests are directed through two layers of caches. Each cache hashes objects to a flash equipped server.

Device	Model A	Model B	Model C
Capacity	670GiB	150GiB	~1.8TiB
Interface	PCI-E	SATA	PCI-E
Seq Write Perf	590MiB/s	160MiB/s	970MiB/s
Rand Write Perf	76MiB/s	19MiB/s	140MiB/s
Read Perf	790MiB/s	260MiB/s	1500MiB/s
Max-Throughput Write Size	512MiB	256MiB	512MiB

Table 1: Flash performance summary. Read and write sizes are 128KiB. Max-Throughput Write Size is the smallest power-of-2 size that achieves sustained maximum throughput at maximum capacity.

caching stack: the Origin cache co-located with backend storage, and the Edge cache spread across the world directly serving photos to the users. Our evaluation shows that both families of algorithms achieve substantially higher hit ratios with RIPQ and SIPQ. For example, GDSF algorithms with RIPQ increase hit ratio in the Origin cache by 17-18%, resulting in a 23-28% decrease in I/O load of the backend.

The contributions of this paper include:

- A flash performance study that identifies a significant increase in the minimum size for max-throughput random writes and motivates the design of RIPQ.
- The design and implementation of RIPQ, our primary contribution. RIPQ is a framework for implementing advanced caching algorithms on flash with high space utilization, high throughput, and long device lifespan.
- The design and implementation of SIPQ, an upgrade from FIFO in memory constrained environments.
- An evaluation on Facebook photo traces that demonstrates advanced caching algorithms on RIPQ (and LRU on SIPQ) can be implemented with high fidelity, high throughput, and low device overhead.

2 Background & Motivation

Facebook's photo-serving stack, shown in Figure 1, includes two caching layers: an Edge cache layer and an Origin cache. At each cache site, individual photo objects are hashed to different caching machines according to their URI. Each caching machine then functions as an independent cache for its subset of objects.[1]

The *Edge cache layer* includes many independent caches spread around the globe at Internet Points of Presence (POP). The main objective of the Edge caching layer—in addition to decreasing latency for users—is decreasing the traffic sent to Facebook's datacenters, so the metric for evaluating its effectiveness is byte-wise hit ratio. The *Origin cache* is a single cache distributed across Facebook's datacenters that sits behind the Edge cache. Its main objective is decreasing requests to Facebook's disk-based storage backends, so the metric for its effectiveness is object-wise hit ratio. Facing high request rates for a large set of objects, both the Edge and Origin caches are equipped with flash drives.

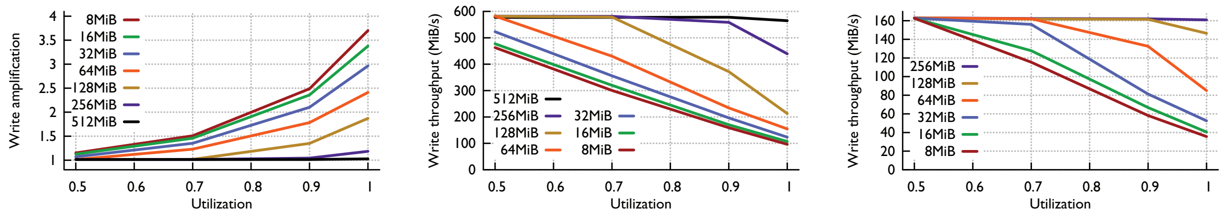
This work is motivated by the finding that SLRU, an advanced caching algorithm, can increase the byte-wise and object-wise hit ratios in the Facebook stack by up to 14% [20]. However, two factors confound naive implementations of advanced caching algorithm on flash. First, the best algorithm for workloads at different cache sites varies. For example, since Huang et al. [20], we have found that GDSF achieves an even higher object-wise hit ratio than SLRU in the Origin cache by favoring smaller objects (see Section 6.2), but SLRU still achieves the highest byte-wise hit ratio at the Edge cache. Therefore, a unified framework for many caching algorithms can greatly reduce the engineering effort and hasten the deployment of new caching policies. Second, flash-based hardware has unique performance characteristics that often require software customization. In particular, a naive implementation of advanced caching algorithms may generate a large number of small random writes on flash, by inserting missed content or updating hit content. The next section demonstrates that modern flash devices perform poorly under such workloads.

3 Flash Performance Study

This section presents a study of modern flash devices that motivates our designs. The study focuses on write workloads that stress the FTL on the devices because write throughput was the bottleneck that prevented Facebook from deploying advanced caching algorithms. Even for a read-only cache, writes are a significant part of the workload as missed content is inserted with a write. At Facebook, even with the benefits of advanced caching algorithms, the maximum hit ratio is ~70%, which results in at least 30% of accesses being writes.

Previous studies [17, 36] have shown that small random writes are harmful for flash. In particular, Min et

[1]Though the stack was originally designed to serve photos, now it handles videos, attachments, and other static binary objects as well. We use "objects" to refer to all targets of the cache in the text.



(a) Write amplification for Model A **(b) Throughput for Model A** **(c) Throughput for Model B**

Figure 2: Random write experiment on Model A and Model B.

al. [36] shows that at high space utilization, i.e., 90%, random write size must be larger than 16 MB or 32 MB to reach peak throughput on three representative SSDs in 2012, with capacities ranging between 32 GB and 64 GB. To update our understanding to current flash devices, we study the performance characteristics on three flash cards, and their specifications and major metrics are listed in Table 1. All three devices are recent models from major vendors,[2] and A and C are currently deployed in Facebook photo caches.

3.1 Random Write Experiments

This subsection presents experiments that explore the trade-off space between write size and device over-provisioning on random write performance. In these experiments we used different sizes to partition the device and then perform aligned random writes of that size under varying space utilizations. We use the flash drive as a raw block device to avoid filesystem overheads. Before each run we use `blkdiscard` to clear the existing data, and then repeatedly pick a random aligned location to perform write/overwrite. We write to the device with 4 times the data of its total capacity before reporting the final stabilized throughput. In each experiment, the initial throughput is always high, but as the device becomes full, the garbage collector kicks in, causing FTL write amplification and dramatic drop in throughput.

During garbage collection, the FTL often writes more data to the physical device than what is issued by the host, and the byte-wise ratio between these two write sizes is the *FTL write amplification* [19]. Figure 2a and Figure 2b show the FTL write amplification and device throughput for the random write experiments conducted on the flash drive Model A. The figures illustrate that as writes become smaller or space utilization increases, write throughput dramatically decreases and FTL write amplification increases. For example, 8 MiB random writes at 90% device utilization achieve only 160 MiB/s, a ~3.7x reduction from the maximum 590 MiB/s. We also experimented with mixed read-write workloads and the same performance trend holds. Specifically, with a 50% read and 50% write workload, 8 MiB random writes at 90% utilization lead to a ~2.3x throughput reduction. High FTL write amplification also reduces device lifespan, and as the erasure cycle continues to decrease for large capacity flash cards, the effects of small random writes become worse over time [5, 39].

Similar throughput results on flash drive Model B are shown in Figure 2c. However, its FTL write amplification is not available due to the lack of monitoring tools for physical writes on the device. Our experiments on flash drive Model C (details elided due to space limitations) agree with Model A and B results as well. Because of the low throughput under high utilization with small write size, more than 1000 device hours are spent in total to produce the data points in Figure 2.

While our findings agree with the previous study [36] in general, we are surprised to find that under 90% device utilization, the minimum write size to achieve peak random write throughput has reached 256 MiB to 512 MiB. This large write size is necessary because modern flash hardware consists of many parallel NAND flash chips [3] and the aggregated erase block size across all parallel chips can add up to hundreds of megabytes. Communications with vendor engineers confirmed this hypothesis. This constraint informs RIPQ's design, which only issues large aligned writes to achieve low write amplification and high throughput.

3.2 Sequential Write Experiment

A common method to achieve sustained high write throughput on flash is to issue sequential writes. The FTL can effectively aggregate sequential writes to parallel erase blocks [30], and on deletes and overwrites all the parallel blocks can be erased together without writing back any still-valid data. As a result, the FTL write amplification can be low or even avoided entirely. To confirm this, we also performed sequential write experiments to the same three flash devices. We observed sustained high performance for all write sizes above 128KiB as reported in Table 1.[3] This result motivates the design of SIPQ, which only issues sequential writes.

[2] Vendor/model omitted due to confidentiality agreements.

[3] Write amplification is low for tiny sequential writes, but they attain lower throughput as they are bound by IOPS instead of bandwidth.

4 RIPQ

This section describes the design and implementation of the RIPQ framework. We show how it approximates the priority queue abstraction on flash devices, present its implementation details, and then demonstrate that it efficiently supports advanced caching algorithms.

4.1 Priority Queue Abstraction

Our experience and previous studies [10, 45] have shown that a *Priority Queue* is a general abstraction that naturally supports various advanced caching policies. RIPQ provides that abstraction by maintaining content in its internal approximate priority queue, and allowing cache operations through three primitives:

- insert(x, p): insert a new object x with priority value p.
- increase(x, p): increase the priority value of x to p.
- delete-min$()$: delete the object with the lowest priority.

The priority value of an object represents its utility to the caching algorithm. On a hit, *increase* is called to adjust the priority of the accessed object. As the name suggests, RIPQ limits priority adjustment to increase only. This constraint simplifies the design of RIPQ and still allows almost all caching algorithms to be implemented. On a miss, *insert* is called to add the accessed object. *Delete-min* is implicitly called to remove the object with the minimum priority value when a cache eviction is triggered by insertion. Figure 3 shows the architecture of a caching solution implemented with the priority queue abstraction, where RIPQ's components are highlighted in gray. These components are crucial to avoid a small-random-writes workload, which can be generated by a naive implementation of priority queue. RIPQ's internal mechanisms are further discussed in Section 4.2.

Absolute/Relative Priority Queue Cache designers using RIPQ can specify the priority of their content based on access time, access frequency, size, and many other factors depending on the caching policy. Although traditional priority queues typically use *absolute* priority values that remain fixed over time, RIPQ operates on a different *relative* priority value interface. In a relative priority queue, an object's priority is a number in the $[0,1]$ range representing the position of the object relative to the rest of the queue. For example, if an object i has a relative priority of 0.2, then 20% of the objects in queue have lower priority values than i and their positions are closer to the tail.

The relative priority of an object is explicitly changed when *increase* is called on it. The relative priority of an object is also *implicitly decreased* as other objects are inserted closer to the head of the queue. For instance, if an object j is inserted with a priority of 0.3, then all objects with priorities ≤ 0.3 will be pushed towards the tail and their priority value implicitly decreased.

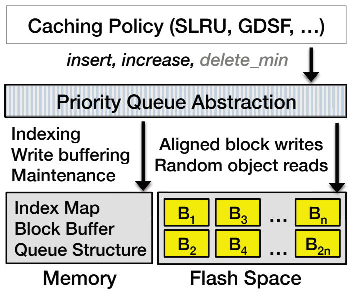


Figure 3: Advanced caching policies with RIPQ.

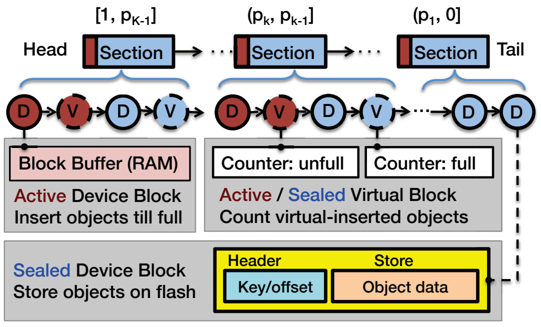


Figure 4: Overall structure of RIPQ.

Many algorithms, including the SLRU family, can be easily implemented with the relative priority queue interface. Others, including the GDSF family, require an absolute priority interface. To support these algorithms RIPQ translates from absolutes priorities to relative priorities, as we explain in Section 4.3.

4.2 Overall Design

RIPQ is a framework that converts priority queue operations into a flash-friendly workload with large writes. Figure 4 gives a detailed illustration of the RIPQ components highlighted in Figure 3, excluding the *Index Map*.

Index Map The *Index Map* is an in-memory hash table which associates all objects' keys with their metadata, including their locations in RAM or flash, sizes, and block IDs. The block structure is explained next.

In our system each entry is ~20 bytes, and RIPQ adds 2 bytes to store the *virtual block ID* of an object. Considering the capacity of the flash card and the average object size, there are about 50 million objects in one caching machine and the index is ~1GiB in total.

Queue Structure The major *Queue Structure* of RIPQ is composed of K sections that are in turn composed of blocks. *Sections* define the insertion points into the queue and a *block* is the unit of data written to flash. The relative priority value range is split

Algorithm	**Interface Used**	**On Miss**	**On Hit**
Segmented-*L* LRU	Relative Priority Queue	$\texttt{insert}(x, \frac{1}{L})$	$\texttt{increase}(x, \frac{\min(1,(1+\lceil p \cdot L \rceil))}{L}))$
Greedy-Dual-Size-Frequency *L*	Absolute Priority Queue	$\texttt{insert}(x, \text{Lowest} + \frac{c(x)}{s(x)})$	$\texttt{increase}(x, \text{Lowest} + c(x)\frac{\min(L,n(x))}{s(x)})$

Table 2: SLRU and GDSF with the priority queue interface provided by RIPQ.

into the K intervals corresponding to the sections: $[1, p_{K-1}], \ldots, (p_k, p_{k-1}], \ldots, (p_1, 0]$.[4] When an object is inserted into the queue with priority p, it is placed in the head of the section whose range contains p. For example, in a queue with sections corresponding to $[1, 0.7]$, $(0.7, 0.3]$ and $(0.1, 0]$, an object with priority value 0.5 would be inserted to the head of second section. Similar to relative priority queues, when an object is inserted to a queue of N objects, any object in the same or lower sections with priority q is *implicitly demoted* from priority q to $\frac{qN}{N+1}$. Implicit demotion captures the dynamics of many caching algorithms, including SLRU and GDSF: as new objects are inserted to the queue, the priority of an old object gradually decreases and it is eventually evicted from the cache when its priority reaches 0.

RIPQ **approximates** the priority queue abstraction because its design restricts where data can be inserted. The insertion point count, K, represents the key design tradeoff in RIPQ between insertion accuracy and memory consumption. Each section has size $O(\frac{1}{K})$, so larger Ks result in smaller sections and thus higher insertion accuracy. However, because each active block is buffered in RAM until it is full and flushed to flash, the memory consumption of RIPQ is proportional to K. Our experiments show $K = 8$ ensures that that RIPQ achieves hit ratios similar to the exact algorithm, and we use this value in our experiments. With 256MiB device blocks, it translates to a moderate memory footprint of 2GiB.

Device and Virtual Blocks As shown in Figure 4, each section includes one active device block, one active virtual block, and an ordered list of sealed device/virtual blocks. An *active device block* accepts insertions of new objects and buffers them in memory, i.e, the *Block Buffer*. When full it is sealed, flushed to flash, and transitions into a *sealed device block*. To avoid duplicating data on flash RIPQ **lazily updates** the location of an object when its priority is increased, and uses *virtual blocks* to track where an object would have been moved. The *active virtual block* at the head of each section accepts *virtually-updated* objects with increased priorities. When the active device block for a section is sealed, RIPQ also transitions the active virtual block into a *sealed virtual block*. Virtual update is an in-memory only operation, which sets the virtual block ID for the object in the *Index Map*, increases the size counter for the target virtual block, and decreases the size counter of the object's original block.

All objects associated with a *sealed device block* are stored in a contiguous space on flash. Within each block, a header records all object keys and their offsets in the data following the header. As mentioned earlier, an updated object is marked with its target virtual block ID within the *Index Map*. Upon eviction of a sealed device block, the block header is examined to determine all objects in the block. The objects are looked up in the *Index Map* to see if their virtual block ID is set, i.e., their priority was increased after insertion. If so, RIPQ *reinserts* the objects to the priorities represented by their virtual blocks. The objects move into active device blocks and their corresponding virtual objects are deleted. Because the updated object will not be written until the old object is about to be evicted, RIPQ maintains at most one copy of each object and duplication is avoided. In addition, lazy updates also allow RIPQ to coalesce all the priority updates to an object between its insertion and reinsertion.

Device blocks occupy a large buffer in RAM (active) or a large contiguous space on flash (sealed). In contrast, virtual blocks resides only in memory and are very small. Each virtual block includes only metadata, e.g., its unique ID, the count of objects in it, and the total byte size of those objects.

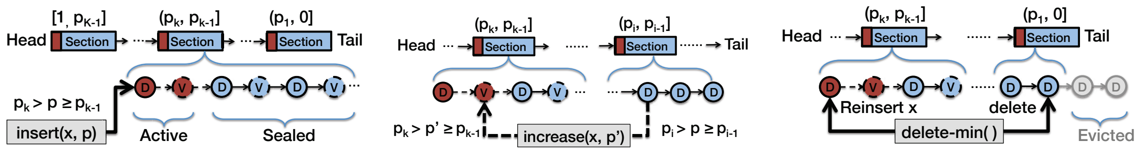
Naive Design One naive design of a priority queue on flash would be to fix an object's location on flash until it is evicted. This design avoids any writes to flash on priority update but does not align the location of an object with its priority. As a result the space of evicted objects on flash would be non-contiguous and the FTL would have to coalesce the scattered objects by copying them forward to reuse the space, resulting in significant FTL write amplification. RIPQ avoids this issue by grouping objects of similar priorities into large blocks and performing writes and evictions on the block level, and by using lazy updates to avoid writes on update.

4.3 Implementing Caching Algorithms

To demonstrate the flexibility of RIPQ, we implemented two families of advanced caching algorithms for evaluation: Segmented-LRU [26], and Greedy-Dual-Size-Frequency [12], both of which yield major caching performance improvement for Facebook photo workload. A summary of the implementation is shown in Table 2.

Segmented-LRU Segmented-*L* LRU (S-*L*-LRU) maintains *L* LRU caches of equal size. On a miss, an

[4] We have inverted the notation of intervals from `[low,high)` to `(high,low]` to make it consistent with the priority order in the figures.



(a) Insertion (b) Increase (c) Delete-min

Figure 5: Insertion, and increase, and delete-min operations in RIPQ.

object is inserted to the head of the L-th (the last) LRU cache. On a hit, an object is promoted to the head of the previous LRU cache, i.e., if it is in sub-cache l, it will be promoted to the head of the $\max(l-1,1)$-th LRU cache. An object evicted from the l-th cache will go to the head of the $(l+1)$-th cache, and objects evicted from the last cache are evicted from the whole cache. This algorithm was demonstrated to provide significant cache hit ratio improvements for the Facebook Edge and Origin caches [20].

Implementing this family of caching algorithms is straightforward with the relative priority queue interface. On a miss, the object is inserted with priority value $\frac{1}{L}$, equaling to the head of the L-th cache. On a hit, based on the existing priority p of the accessed object, RIPQ promotes it from the $\lceil (1-p)\cdot L \rceil$-th cache to the head of the previous cache with the new, higher priority $\min(1, \frac{1+\lceil p\cdot L \rceil}{L})$. With the relative priority queue abstraction, an object's priority is automatically decreased when another object is inserted/updated to a higher priority. When an object is inserted at the head of the l-th LRU cache, all objects in l-th to L-th caches are demoted, and ones at the tail of these caches will be either demoted to the next lower priority cache or evicted if they are in the last L-th cache—the dynamics of SLRU are exactly captured by relative priority queue interface.

Greedy-Dual-Size-Frequency The Greedy-Dual-Size algorithm [10] provides a principled way to trade-off increased object-wise hit ratio with decreased byte-wise hit ratio by favoring smaller objects. It achieves an even higher object-wise hit ratio for the Origin cache than SLRU (Section 2), and is favored for that use case as the main purpose of Origin cache is to protect backend storage from excessive IO requests. Greedy-Dual-Size-Frequency [12] (GDSF) improves GDS by taking frequency into consideration. In GDSF, we update the priority of an object x to be $\texttt{Lowest} + c(x) \cdot \frac{n}{s(x)}$ upon its n-th access since it was inserted to the cache, where $c(x)$ is the programmer-defined penalty for a miss on x, `Lowest` is the lowest priority value in the current priority queue, and $s(x)$ is the size of the object. We use a variant of GDSF that caps the maximum value of the frequency of an object to L. L is similar to the number of segments in SLRU. It prevents the priority value of a frequently accessed object from blowing up and adapts better to dynamic workloads. The update rule of our variant of GDSF algorithm is thus $p(x) \leftarrow \texttt{Lowest} + c(x) \cdot \frac{\min(L,n)}{s(x)}$. Because we are maximizing object-wise hit ratio we set $c(x) = 1$ for all objects. GDSF uses the absolute priority queue interface.

Limitations RIPQ also supports many other advanced caching algorithms like LFU, LRFU [28], LRU-k [40], LIRS [24], SIZE [1], but there are a few notable exceptions that are not implementable with a single RIPQ, e.g., MQ [48] and ARC [34]. These algorithms involve multiple queues and thus cannot be implemented with one RIPQ. Extending our design to support them with multiple RIPQs coexisting on the same hardware is one of our future directions. A harder limitation comes from the update interface, which only allows increasing priority values. Algorithms that decrease the priority of an object on its access, such as MRU [13], cannot be implemented with RIPQ. MRU was designed to cope with scans over large data sets and does not apply to our use case.

RIPQ does not support delete/overwrite operation because such operations are not needed for static content such as photos. But, they are necessary for a general-purpose read-write cache and adding support for them is also one of our future directions.

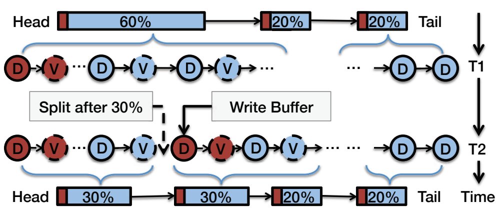
4.4 Implementation of Basic Operations

RIPQ implements the three operations of a regular priority queue with the data structures described above.

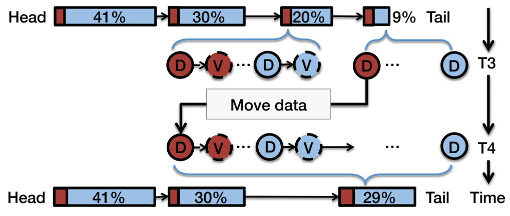
`Insert`(x,p) RIPQ inserts the object to the active device block of section k that contains p, i.e., $p_k > p \geq p_{k-1}$.[5] The write will be buffered until that active block is sealed. Figure 5a shows an insertion.

`Increase`(x,p) RIPQ avoids moving object x that is already resident in a device block in the queue. Instead, RIPQ virtually inserts x into the active virtual block of section k that contains p, i.e., $p_k > p \geq p_{k-1}$, and logically removes it from its current location. Because we remember the virtual block ID in the object entry in the indexing hash table, these steps are simply implemented by setting/resetting the virtual block ID of the object entry, and updating the size counters of the blocks and sec-

[5] A minor modification when $k = K$ is $1 = p_k \geq p \geq p_{k-1}$.



(a) Section split process



(b) Section merge process

Figure 6: RIPQ internal operations.

tions accordingly. No read/write to flash is performed during this operation. Figure 5b shows an update.

`Delete-min()` We maintain a few reserved blocks on flash for flushing the RAM buffers of device blocks when they are sealed.[6] When the number of reserved blocks falls below this threshold, the `Delete-min()` operation is called implicitly to free up the space on flash. As shown in Figure 5c, the lowest-priority block in queue is evicted from queue during the operation. However, because some of the objects in that blocks might have been updated to higher places in the queue, they need to be reinserted to maintain their correct priorities. The reinsertion (1) reads out all the keys of the objects in that block from the block header, (2) queries the index structure to find whether an object, x, has a virtual location, and if it has one, (3) finds the corresponding section, k, of that virtual block and copies the data to the active device block of that section in RAM, and (4) finally sets the virtual block field in the index entry to be empty. We call this whole process *materialization* of the virtual update.

These reinsertions help preserve caching algorithm fidelity, but cause additional writes to flash. These additional writes cause *implementation write amplification*, which is the byte-wise ratio of host-issued writes to those required to inserted cache misses. RIPQ can explicitly trade lower caching algorithm fidelity for lower write amplification by skipping materialization of the virtual objects whose priority is smaller than a given threshold, e.g., in the last 5% of the queue. This threshold is the *logical occupancy parameter* θ $(0 < \theta < 1)$.

Internal operations RIPQ must have neither too many nor too few insertion points: too few leads to low accuracy, and too many leads to high memory usage. To avoid these situations RIPQ splits a section when it grows too large and merges consecutive sections when their total size is too small. This is similar to how B-tree [7] splits/merges nodes to control the size of the nodes and the depth of the tree.

A parameter α controls the number of sections of RIPQ in a principled way. α is in $(0,1)$ and determines the average size of sections. RIPQ splits a section when its relative size—i.e., a ratio based on the object count or byte size—has reached 2α. For example, if $\alpha = 0.3$ then a section of $[0.4, 1.0]$ would be split to two sections of $[0.4, 0.7)$ and $[0.7, 1.0]$ respectively, shown in Figure 6a. RIPQ merges two consecutive sections if the sum of their sizes is smaller than α, shown in Figure 6b. These operations ensure there are at most $\lceil \frac{2}{\alpha} \rceil$ sections, and that each section is no larger than 2α.

No data is moved on flash for a split or merge. Splitting a section creates a new active device block with a write buffer and a new active virtual block. Merging two sections combines their two active device blocks: the write buffer of one is copied into the write buffer of the other. Splitting happens often and is how new sections are added to queue as objects in the section at the tail are evicted block-by-block. Merging is rare because it requires the total size of two consecutive sections to shrink from 2α (α is the size of a new section after a split) to α to trigger a merge. The amortized complexity of a merge per operation provided by the priority queue API is only $O(\frac{1}{\alpha M})$, where M is the number of blocks.

Supporting Absolute Priorities Caching algorithms such as LFU, SIZE [1], and Greedy-Dual-Size[10] require the use of absolute priority values when performing insertion and update. RIPQ supports absolute priorities with a mapping data structure that translates them to relative priorities. The data structure maintains a dynamic histogram that supports insertion/deletion of absolute priority values, and when given an absolute priorities return approximate quantiles, which are used as the internal relative priority values.

The histogram consists of a set of bins, and we merge/split bins dynamically based on their relative sizes, similar to the way we merge/split sections in RIPQ. We can afford to use more bins than sections for this dynamic histogram and achieve higher accuracy of the translation, e.g., $\kappa = 100$ bins while RIPQ only uses $K = 8$ sections, because the bins only contains absolute priority values and do not require a large dedicated RAM buffer as the sections do. Consistent sampling of keys to insert priority values to the histogram can be fur-

[6]It is not a critical parameter and we used 10 in our evaluation.

Parameter	Symbol	Our Value	Description and Goal
Block Size	B	256MiB	To satisfy the sustained high random write throughput.
Number of Blocks	M	2400	Flash caching capacity divided by the block size.
Average Section Size	α	0.125	To bound the number of sections $\leq \lceil 2/\alpha \rceil$ and the size of each section $\leq 2\alpha$, trade-off parameter for insertion accuracy and RAM buffer usage.
Insertion Points	K	8	Same as the number of sections, controlled by α and proportional to RAM buffer usage.
Logical Occupancy	θ	0	Avoid reinsertion of items that will soon be permanently evicted.

Table 3: Key parameters of RIPQ for a 670GiB flash drive currently deployed in Facebook.

ther applied to reduce its memory consumption and insertion/update complexity.

4.5 Other Design Considerations

Parameters Table 3 describes the parameters of RIPQ and the value chosen for our implementation. The *block size B* is chosen to surpass the threshold for a sustained high write throughput for random writes, and the *number of blocks M* is calculated directly based on cache capacity. The number of blocks affects the memory consumption of RIPQ, but this is dominated by the size of the write buffers for active blocks and the indexing structure. The number of active blocks equals the number of *insertion points K* in the queue. The *average section size* α is used by the split and merge operations to bound the memory consumption and approximation error of RIPQ.

Durability Durability is not a requirement for our static-content caching use case, but not having to refill the entire cache after a power loss is a plus. Fortunately, because the keys and locations of the objects are stored in the headers of the on-flash device blocks, all objects that have been saved to flash can be recovered, except for those in the RAM buffers. The ordering of blocks/sections can be periodically flushed to flash as well and then used to recover the priorities of the objects.

4.6 Theoretical Analysis

RIPQ is a practical approximate priority queue for implementing caching algorithms on flash, but enjoys some good theoretical properties as well. In the appendix of a longer technical report [44] we show RIPQ can simulate a LRU cache *faithfully* with 4α of additional space: if $\alpha = 0.125$, this would mean RIPQ-based LRU with 50% additional space would provably include all the objects in an exact LRU cache. In general RIPQ with adjusted insertion points can simulate a S-L- LRU cache with $4L\alpha$ of additional space. It is also easy to show the number of writes to the flash is $\leq I + U$, where I is the number of inserts and U is the number of updates.

Using K sections/insertion points, the complexity of finding the approximate insertion/update point takes $O(K)$, and the amortized complexity of split/merge internal operations is $O(1)$, so the amortized complexity of RIPQ is only $O(K)$. If we arrange the sections in a red-black tree, it can be further reduced to $O(\log K)$. In comparison to this, with N objects, an exact implementation of priority queues using red-black tree would take $O(\log N)$ per operation, and a Fibonacci heap takes $O(\log N)$ per delete-min operation. ($K \ll N$, K is typically 8, N is typically 50 million). The computational complexity of these exact, tree and heap based data structures are not ideal for a high performance system. In contrast, RIPQ hits the sweet spot with fast operations and high fidelity, in terms of both theoretical analysis and empirical hit ratios.

5 SIPQ

RIPQ's buffering for large writes creates a moderate memory footprint, e.g., 2 GiB DRAM for 8 insertion points with 256 MiB block size in our implementation. This is not an issue for servers at Facebook, which are equipped with 144 GiB of RAM, but limits the use of RIPQ in memory-constrained environments. To cope with this issue, we propose the simpler Single Insertion Priority Queue (SIPQ) framework.

SIPQ uses flash as a cyclic queue and only sequentially writes to the device for high write throughput with minimal buffering. When the cache is full, SIPQ reclaims device space following the same sequential order. In contrast to RIPQ, SIPQ maintains an exact priority queue of the *keys* of the cached objects in memory and does not co-locate similarly prioritized objects physically due to the single insertion limit on flash. The drawback of this approach is that reclaiming device space may incur many reinsertions for SIPQ in order to preserve its priority accuracy. Similar to RIPQ, these reinsertions constitute the implementation write amplification of SIPQ.

To reduce the implementation write amplification, SIPQ only includes the keys of a portion of all the cached objects in the in-memory priority queue, referred to as the *virtual cache*, and will only reinsert evicted objects that are in this cache. All on-flash capacity is referred to as the *physical cache* and the ratio between the total byte size of objects in the virtual cache to the size of the physical cache is controlled by a *logical occupancy parameter* θ ($0 < \theta < 1$). Because only objects in the virtual cache are reinserted when they are about to be evicted from the physical cache, θ provides a trade-off between priority fidelity and implementation write amplification: the larger θ, the more objects are in the virtual cache and the higher fidelity SIPQ has relative to the exact caching algorithm, and on the other hand the more likely evicted objects will need to be reinserted and thus higher write amplification caused by SIPQ. For $\theta = 1$, SIPQ implements an exact priority queue for all cached data on flash, but incurs high write amplification for reinsertions. For $\theta = 0$, SIPQ deteriorates to FIFO with no priority enforcement. For θ in between, SIPQ performs additional writes compared to FIFO but also delivers part of the improvement of more advanced caching algorithms. In our evaluation, we find that SIPQ provides a good trade-off point for Segmented-LRU algorithms with $\theta = 0.5$, but does not perform well for more complex algorithms like GDSF. Therefore, with limited improvement at almost no additional device overhead, SIPQ can serve as a simple upgrade for FIFO when memory is tight.

6 Evaluation

We compare RIPQ, SIPQ, and Facebook's current solution, FIFO, to answer three key questions:

1. What is the impact of RIPQ and SIPQ's approximations of caching algorithms on hit ratios, i.e., what is the effect on algorithm fidelity?
2. What is the write amplification caused by RIPQ and SIPQ versus FIFO?
3. What throughput can RIPQ and SIPQ achieve?
4. How does the hit-ratio of RIPQ change as we vary the number of insertion points?

6.1 Experimental Setup

Implementation We implemented RIPQ and SIPQ with 1600 and 600 lines of C++ code, respectively, using the Intel TBB library [22] for the object index and the C++11 thread library [9] for the concurrency mechanisms. Both the relative and absolute priority interfaces (enabled by an adaptive histogram translation) are supported in our prototypes.

Hardware Environment Experiments are run on servers equipped with a Model A 670GiB flash device and 144GiB DRAM space. All flash devices are configured with 90% space utilization, leaving the remaining 10% for the FTL.

Framework Parameters RIPQ uses a 256MiB block size to achieve high write throughput based on our performance study of Model A flash in Section 3. It uses $\alpha = 0.125$, i.e., 8 sections, to provide a good trade-off between the fidelity to the implemented algorithms and the total DRAM space RIPQ uses for buffering: 256MiB $\times$ 8 = 2GiB, which is moderate for a typical server.

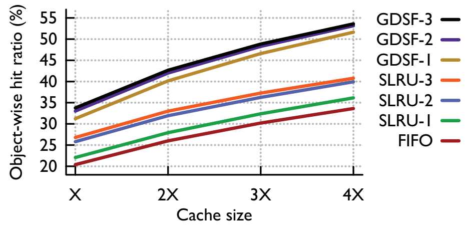
SIPQ also uses the 256MiB block size to keep the number of blocks on flash the same as RIPQ. Because SIPQ only issues sequential writes, its buffering size could be further shrunk without adverse effects. Two *logical occupancy* values for SIPQ are used in evaluation: 0.5, and 0.9, each representing a different trade-off between the approximation fidelity to the exact algorithm and implementation write amplification. These two settings are noted as SIPQ-0.5 and SIPQ-0.9, respectively.

Caching Algorithms Two families of advanced caching algorithms, Segmented-LRU (SLRU) [26] and Greedy-Dual-Size-Frequency (GDSF) [12], are evaluated on RIPQ and SIPQ. For Segmented-LRU, we vary the number of segments from 1 to 3, and report their results as SLRU-1, SLRU-2, and SLRU-3, respectively. We similarly set L from 1 to 3 for Greedy-Dual-Size-Frequency, denoted as GDSF-1, GDSF-2, and GDSF-3. Description of these algorithms and their implementations on top of the priority queue interface are explained in Section 4.3. Results of 4 segments or more for SLRU and $L \geq 4$ for GDSF are not included due to their marginal differences in the caching performance.

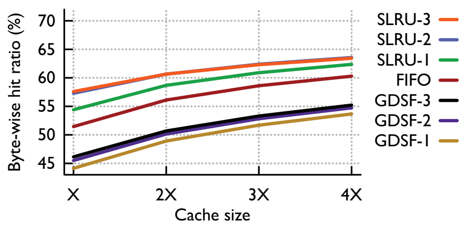
Facebook Photo Trace Two sets of 15-day sampled traces collected within the Facebook photo-serving stack are used for evaluation, one from the Origin cache, and the other from a large Edge cache facility. The Origin trace contains over 4 billion requests and 100TB worth of data, and the Edge trace contains over 600 million requests and 26TB worth of data. To emulate different total cache capacities in Origin/Edge with the same space utilization of the experiment device and thus controlling for the effect of FTL, both traces are further down sampled through hashing: we randomly sample $\frac{1}{2}$, $\frac{1}{3}$, and $\frac{1}{4}$ of the cache key space of the original trace for each experiment to emulate the effect of increasing the total caching capacity to $2X$, $3X$, and $4X$. We report experimental results at $2X$ because it closely matches our production configurations. For all evaluation runs, we use the first 10-day trace to warm up the cache and measure performance during the next 5 days. Because both the working set and the cache size are very large, it takes hours to fill up the cache and days for the hit ratio to stabilize.

6.2 Experimental Results

This section presents our experimental results regarding the algorithm fidelity, write amplification, and throughput of RIPQ and SIPQ with the Facebook photo trace. We also include the hit ratio, write amplification and throughput achieved by Facebook's existing FIFO solution as a baseline. For different cache sites, only their target hit ratio metrics are reported, i.e., object-wise hit ratio for the Origin trace and byte-wise hit ratio for the Edge trace. Exact algorithm hit ratios are obtained via simulations as the baseline to judge the approximation fidelity of implementations on top of RIPQ and SIPQ.



(a) **Object-wise hit ratios on Origin trace.**



(b) **Byte-wise hit ratios on Edge trace.**

Figure 7: Exact algorithm hit ratios on Facebook trace.

Performance of Exact Algorithms We first investigate hit ratios achieved by the exact caching algorithms to determine the gains of a fully accurate implementation. Results are shown in Figure 7.

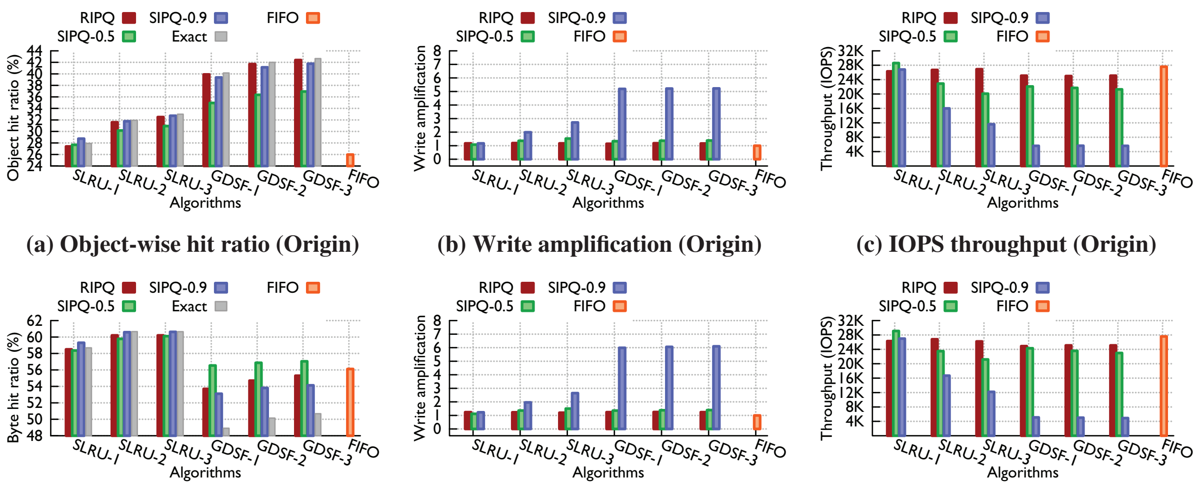
For object-wise hit ratio on the Origin trace, Figure 7a shows that GDSF family outperforms SLRU and FIFO by a large margin. At $2X$ cache size, GDSF-3 increases the hit ratio over FIFO by 17%, which translates a to a 23% reduction of backend IOPS. For byte-wise hit ratio on the Edge trace, Figure 7b shows that SLRU is the best option: at $2X$ cache size, SLRU-3 improves the hit ratio over FIFO by 4.5%, which results in a bandwidth reduction between Edge and Origin by 10%. GDSF performs poorly on the byte-wise metric because it down weights large photos. Because different algorithms perform best at different sites with different performance metrics, flexible frameworks such as RIPQ make it easy to optimize caching policies with minimal engineering effort.

Approximation Fidelity Exact algorithms yield considerable gains in our simulation, but are also challenging to implement on flash. RIPQ and SIPQ make it simple to implement the algorithms on flash, but do so by approximating the algorithms. To quantify the effects of this approximation we ran experiments presented in Figures 8a and 8d. These figures present the hit ratios of different exact algorithms (in simulations) and their approximate implementations on flash with RIPQ, SIPQ-0.5, and SIPQ-0.9 (in experiments) at $2X$ cache size setup from Figure 7. The implementation of FIFO is the same as the exact algorithm, so we only report one number. In general, if the hit ratio of an implementation is similar to the exact algorithm the framework provides high fidelity.

RIPQ consistently achieves high approximation fidelities for the SLRU family, and its hit ratios are less than 0.2% different for object-wise/byte-wise metric compared to the exact algorithm results on Origin/Edge trace. For the GDSF family, RIPQ's algorithm fidelity becomes lower as the algorithm complexity increases. The greatest "infidelity" seen for RIPQ is a 5% difference on the Edge trace for GDSF-1. Interestingly, for the GDSF family, the infidelity generated by RIPQ improves byte-wise hit ratio—the largest infidelity was a 5% improvement on byte-wise hit-ratio compared to the exact algorithm. The large gain on byte-wise hit ratio can be explained by the fact that the exact GDSF algorithm is designed to trade byte-wise hit ratio for object-wise hit ratio through favoring small objects, and its RIPQ approximation shifts this trade-off back towards a better byte-wise hit-ratio. Not shown in the figures (due to space limitation) is that RIPQ-based GDSF family incurs about 1% reduction in *object-wise* hit ratio. Overall, RIPQ achieves high algorithm fidelity on both families of caching algorithms that perform the best in our evaluation.

SIPQ also has high fidelity when the occupancy parameter is set to 0.9, which means 90% of the caching capacity is managed by the exact algorithm. SIPQ-0.5, despite only half of the cache capacity being managed by the exact algorithm, still achieves a relatively high fidelity for SLRU algorithms: it creates a 0.24%-2.8% object-wise hit ratio reduction on Origin, and 0.3%-0.9% byte-wise hit ratio reduction on Edge. These algorithms tend to put new and recently accessed objects towards the head of the queue, which is similar to the way SIPQ inserts and reinserts objects at the head of the cyclic queue on flash. However, SIPQ-0.5 provides low fidelity for the GDSF family, causing object-wise hit ratio to decrease on Origin and byte-wise hit ratio to increase on Edge. Within these algorithms, objects may have diverse priority values due to their size differences even if they enter the cache at the same time, and SIPQ's single insertion point design results in a poor approximation.

Write Amplification Figure 8b and 8e further show the combined write amplification (i.e., $FTL \times implementation$) of different frameworks. RIPQ consistently achieves the lowest write amplification, with an exception for SLRU-1 where SIPQ-0.5 has the lowest value for both traces. This is because SLRU-1 (LRU) only inserts to one location at the queue head, which works well with SIPQ, and the logical occupancy 0.5 further reduces the reinsertion overhead. Overall, the write amplification of RIPQ is largely stable regardless of the complexity of the caching algorithms, ranging from 1.17 to 1.24 for the SLRU family, and from 1.14 to 1.25 for the GDSF family.



(a) Object-wise hit ratio (Origin) **(b) Write amplification (Origin)** **(c) IOPS throughput (Origin)**

(d) Byte-wise hit ratio (Edge) **(e) Write amplification (Edge)** **(f) IOPS throughput (Edge)**

Figure 8: Performance of RIPQ, SIPQ, and FIFO on Origin and Edge.

SIPQ-0.5 achieves moderately low write amplifications but with lower fidelity for complex algorithms. Its write amplification also increases with the algorithm complexity. For SLRU, the write implementation for SIPQ-0.5 rises from 1.08 for SLRU-1 to 1.52 for SLRU-3 on Origin, and from 1.11 to 1.50 on Edge. For GDSF, the value ranges from 1.33 for GDSF-1 to 1.37 to GDSF-3 on Origin, and from 1.36 to 1.39 on Edge. Results for SIPQ-0.9 observe a similar trend for each family of algorithms, but with a much higher write amplification value for GDSF around 5-6.

Cache Throughput Throughput results are shown in Figure 8c and 8f. RIPQ and SIPQ-0.5 consistently achieve over 20 000 requests per second (rps) on both traces, but SIPQ-0.9 has considerably lower throughput, especially for the GDSF family of algorithms. FIFO has slightly higher throughput than RIPQ based SLRU, although the latter has higher byte hit ratio and correspondingly fewer writes from misses.

This performance is highly related to the write amplification results because in all three frameworks (1) workloads are write-heavy with below 63% hit ratios, and our experiments are mainly write-bounded with a sustained write-throughput around 530 MiB/sec, (2) write amplification proportionally consumes the write throughput, which further throttles the overall throughput. This is why SIPQ-0.9 often with the highest write amplification has the lowest throughput, and also why RIPQ based SLRU has lower throughput than FIFO. However, RIPQ/SIPQ-0.5 still provides high performance for our use case, with RIPQ paticularly achieving over 24 000 rps on both traces. The slightly lower throughput comparing to FIFO (less than 3 000 rps difference) is well worth the hit-ratio improvement which translates to a decrease of backend I/O load and a decrease of bandwidth between Edge and Origin.

Sensitivity Analysis on Number of Insertion Points Figure 9 shows the effect of varying the number of insertion points in RIPQ on approximation accuracy. The number of insertion points, K, is roughly inversely proportional to α, so we vary K to be approximately $2, 4, 8, 16$, and 32, by varying α from $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}$ to $\frac{1}{32}$. We measure approximation accuracy empirically through the object-wise hit-ratios of RIPQ based SLRU-3 and GDSF-3 on the origin trace with 2X cache size.

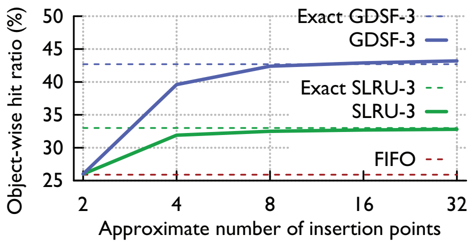


Figure 9: Object-wise hit ratios sensitivity on approximate number of insertion points.

When $K \approx 2$ ($\alpha = \frac{1}{2}$), a section in RIPQ can grow to the size of the entire queue before it splits. In this case RIPQ effectively degenerates to FIFO with equivalent hit-ratios. The SLRU-3 hit ratio saturates quickly when $K \gtrsim 4$, while GDSF-3 reaches its highest performance only when $K \gtrsim 8$. GDSF-3 uses many more insertion points in an exact priority queue than SLRU-3 and RIPQ thus needs more insertion points to effectively colocate content with similar priorities. Based on this analysis we have chosen $\alpha = \frac{1}{8}$ for RIPQ in our experiments.

7 Related Work

To the best of our knowledge, no prior work provides a flexible framework for efficiently implementing advanced caching algorithms on flash. Yet, there is a large body of related work in several heavily-researched fields.

Flash-based Caching Solutions Flash devices have been applied in various caching solutions for their large capacities and high I/O performance [2, 4, 21, 23, 27, 31, 35, 37, 39, 42, 46]. To avoid their poor handling of small random write workloads, previous studies either use sequential eviction akin to FIFO [2], or only perform coarse-grained caching policies at the unit of large blocks [21, 31, 46]. Similarly, SIPQ and RIPQ also achieve high write throughputs and low device overheads on flash through sequential writes and large aligned writes, respectively. In addition, they allow efficient implementations of advanced caching policies at a fine-grained object unit, and our experience show that photo caches built on top of RIPQ and SIPQ yield significant performance gains at Facebook. While our work mainly focuses on the support of *eviction* part of caching operations, techniques like *selective insertions* on misses [21, 46] are orthogonal to RIPQ and can be applied to further reduce the data written to flash.[7]

RAM-based Advanced Caching Caching has been an important research topic since the early days of computer science and many algorithms have been proposed to better capture the characteristics of different workloads. Some well-known features include recency (LRU, MRU [13]), frequency (LFU [33]), inter-reference time (LIRS [24]), and size (SIZE [1]). There have also been a plethora of more advanced algorithms that consider multiple features, such as Multi-Queue [48] and Segmented LRU (SLRU) [26] for both recency and frequency, Greedy-Dual [47] and its variants like Greedy-Dual-Size [10] and Greedy-Dual-Size-Frequency [12] (GDSF) using a more general method to compose the expected miss penalty and minimize it.

While more advanced algorithms can potentially yield significant performance improvements, such as SLRU and GDSF for Facebook photo workload, a gap still remains for efficient implementations on top of flash devices because most algorithms are hardware-agnostic: they implicitly assume data can be moved and overwritten with little overhead. Such assumptions do not hold on flash due to its asymmetric performance for reads and writes and the performance deterioration caused by its internal garbage collection.

Our work, RIPQ and SIPQ, bridges this gap. They provide a priority queue interface to allow easy implementation of many advanced caching algorithms, providing similar caching performance while generating flash-friendly workloads.

Flash-based Store Many flash-based storage systems, especially key-value stores have been recently proposed to work efficiently on flash hardware. Systems such as FAWN-KV [6], SILT [32], LevelDB [16], and RocksDB [14] group write operations from an upper layer and only flush to the device using sequential writes. However, they are designed for read-heavy workloads and other performance/application metrics such as memory footprints and range-query efficiencies. As a result, these systems make trade-offs such as conducting on-flash data sorting and merges, that yield high device overhead for write-heavy workloads. We have experimented with using RocksDB as an on-flash photo store for our application, but found it to have excessively high write amplification (~5 even when we allocated 50% of the flash space to garbage collection). In contrast, RIPQ and SIPQ are specifically optimized for a (random) write-heavy workload and only support caching-required interfaces, and as a result have low write amplification.

Studies on Flash Performance and Interface While flash hardware itself is also an important topic, works that study the application perceived performance and interface are more related to our work. For instance, previous research [8, 25, 36, 43] that reports the random write performance deterioration on flash helps verify our observations in the flash performance study.

Systematic approaches to mitigate this specific problem have also been previously proposed at different levels, such as separating the treatment of cold and hot data in the FTL by LAST [29], and the similar technique in filesystem by SFS [36]. These approaches work well for skewed write workloads where only a small subset of the data is hot and updated often, and thus can be grouped together for garbage collection with lower overhead. In RIPQ, cached contents are explicitly tagged with priority values that indicate their hotness, and are co-located within the same device block if their priority values are close. In a sense, such priorities provide a *prior* for identifying content hotness.

While RIPQ (and SIPQ) runs on unmodified commercial flash hardware, recent studies [31, 41] which co-design flash software/hardware could further benefit RIPQ by reducing its memory consumption.

Priority Queue Both RIPQ and SIPQ rely on the priority queue abstract data type and the design of priority queues with different performance characteristics have been a classic topic in theoretical computer science as well [11, 15, 18]. Instead of building an exact priority queue, RIPQ uses an approximation to trade algorithm fidelity for flash-aware optimization.

[7]We tried such techniques on our traces, but found the hit ratio dropped because of the long-tail accesses for social network photos.

8 Conclusion

Flash memory, with its large capacity, high IOPS, and complex performance characteristics, poses new opportunities and challenges for caching. In this paper we present two frameworks, RIPQ and SIPQ, that implement approximate priority queues efficiently on flash. On top of them, advanced caching algorithms can be easily, flexibly, and efficiently implemented, as we demonstrate for the use case of a flash-based photo cache at Facebook. RIPQ achieves high fidelity and low write amplification for the SLRU and GDSF algorithms. SIPQ is a simpler design, requires less memory and still achieves good results for simple algorithms like LRU. Experiments on both the Facebook Edge and Origin traces show that RIPQ can improve hit ratios by up to ~20% over the current FIFO system, reducing bandwidth consumption between the Edge and Origin, and reducing I/O operations to backend storage.

Acknowledgments We are grateful to our shepherd Philip Shilane, the anonymous reviewers of the FAST program committee, Xin Jin, Xiaozhou Li, Kelvin Zou, Mohsin Ali, Ken Birman, and Robbert van Renesse for their extensive comments that substantially improved this work. We are also grateful to Brian Pane, Bryan Alger, Peter Ruibal, and other Facebook engineers who gave feedback that improved our designs. Our work is supported by Facebook, NSF grant BCS-1229597, Intel, DARPA MRC program, and Princeton fellowship.

References

[1] M. Abrams, C. R. Standridge, G. Abdulla, E. A. Fox, and S. Williams. Removal policies in network caches for World-Wide Web documents. In *ACM SIGCOMM Computer Communication Review*, 1996.

[2] A. Aghayev and P. Desnoyers. Log-structured cache: trading hit-rate for storage performance (and winning) in mobile devices. In *USENIX INFLOW*, 2013.

[3] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In *USENIX ATC*, 2008.

[4] C. Albrecht, A. Merchant, M. Stokely, M. Waliji, F. Labelle, N. Coehlo, X. Shi, and E. Schrock. Janus: Optimal Flash Provisioning for Cloud Storage Workloads. In *USENIX ATC*, 2013.

[5] D. G. Andersen and S. Swanson. Rethinking flash in the data center. *IEEE Micro*, 2010.

[6] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *ACM SOSP*, 2009.

[7] R. Bayer and E. McCreight. *Organization and maintenance of large ordered indexes*. Springer, 2002.

[8] L. Bouganim, B. r Jnsson, and P. Bonnet. uFLIP: Understanding Flash IO Patterns. In *CIDR*, 2009.

[9] C++11 Thread Support Library. http://en.cppreference.com/w/cpp/thread, 2014.

[10] P. Cao and S. Irani. Cost-Aware WWW Proxy Caching Algorithms. In *USENIX USITS*, 1997.

[11] B. Chazelle. The soft heap: an approximate priority queue with optimal error rate. *Journal of the ACM*, 2000.

[12] L. Cherkasova and G. Ciardo. Role of aging, frequency, and size in web cache replacement policies. In *Springer HPCN*, 2001.

[13] H.-T. Chou and D. J. DeWitt. An evaluation of buffer management strategies for relational database systems. *Algorithmica*, 1986.

[14] Facebook Database Engineering Team. RocksDB, A persistent key-value store for fast storage environments. http://rocksdb.org, 2014.

[15] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 1987.

[16] S. Ghemawat and J. Dean. LevelDB, A fast and lightweight key/value database library by Google. https://github.com/google/leveldb, 2014.

[17] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *ACM ASPLOS*, 2009.

[18] J. E. Hopcroft. Data structures and algorithms. *AddisonWeely*, 1983.

[19] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka. Write amplification analysis in flash-based solid state drives. In *ACM SYSTOR*, 2009.

[20] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An Analysis of Facebook Photo Caching. In *ACM SOSP*, 2013.

[21] S. Huang, Q. Wei, J. Chen, C. Chen, and D. Feng. Improving flash-based disk cache with Lazy Adaptive Replacement. In *IEEE MSST*, 2013.

[22] Intel Thread Building Blocks. https://www.threadingbuildingblocks.org, 2014.

[23] D. Jiang, Y. Che, J. Xiong, and X. Ma. uCache: A Utility-Aware Multilevel SSD Cache Management Policy. In *IEEE HPCC_EUC*, 2013.

[24] S. Jiang and X. Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *ACM SIG-*

METRICS, 2002.

[25] K. Kant. Data center evolution: A tutorial on state of the art, issues, and challenges. *Computer Networks*, 2009.

[26] R. Karedla, J. S. Love, and B. G. Wherry. Caching strategies to improve disk system performance. *IEEE Computer*, 1994.

[27] T. Kgil, D. Roberts, and T. Mudge. Improving NAND flash based disk caches. In *ACM/IEEE ISCA*, 2008.

[28] D. Lee, J. Choi, J. H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. LRFU: A Spectrum of Policies That Subsumes the Least Recently Used and Least Frequently Used Policies. *IEEE Transactions on Computers*, 2001.

[29] S. Lee, D. Shin, Y.-J. Kim, and J. Kim. LAST: locality-aware sector translation for NAND flash memory-based storage systems. *ACM SIGOPS Operating Systems Review*, 2008.

[30] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions on Embedded Computing Systems*, 2007.

[31] C. Li, P. Shilane, F. Douglis, H. Shim, S. Smaldone, and G. Wallace. Nitro: A Capacity-Optimized SSD Cache for Primary Storage. In *USENIX ATC*, 2014.

[32] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *ACM SOSP*, 2011.

[33] S. Maffeis. Cache management algorithms for flexible filesystems. *ACM SIGMETRICS Performance Evaluation Review*, 1993.

[34] N. Megiddo and D. S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *USENIX FAST*, 2003.

[35] F. Meng, L. Zhou, X. Ma, S. Uttamchandani, and D. Liu. vCacheShare: Automated Server Flash Cache Space Management in a Virtualization Environment. In *USENIX ATC*, 2014.

[36] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom. SFS: Random write considered harmful in solid state drives. In *USENIX FAST*, 2012.

[37] D. Mituzas. Flashcache at Facebook: From 2010 to 2013 and beyond. `https://www.facebook.com/notes/10151725297413920`, 2014.

[38] Netflix. Netflix Open Connect. `https://www.netflix.com/openconnect`, 2014.

[39] Y. Oh, J. Choi, D. Lee, and S. H. Noh. Improving performance and lifetime of the SSD RAID-based host cache through a log-structured approach. In *USENIX INFLOW*, 2013.

[40] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *ACM SIGMOD*, 1993.

[41] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang. SDF: software-defined flash for web-scale internet storage systems. In *ACM ASPLOS*, 2014.

[42] M. Saxena, M. M. Swift, and Y. Zhang. Flashtier: a lightweight, consistent and durable storage cache. In *ACM EuroSys*, 2012.

[43] R. Stoica, M. Athanassoulis, R. Johnson, and A. Ailamaki. Evaluating and repairing write performance on flash devices. In *ACM DAMON*, 2009.

[44] L. Tang, Q. Huang, W. Lloyd, S. Kumar, and K. Li. RIPQ Princeton Technical Report. `https://www.cs.princeton.edu/research/techreps/TR-977-15`, 2014.

[45] R. P. Wooster and M. Abrams. Proxy caching that estimates page load delays. *Computer Networks and ISDN Systems*, 1997.

[46] J. Yang, N. Plasson, G. Gillis, and N. Talagala. Hec: improving endurance of high performance flash-based cache devices. In *ACM SYSTOR*, 2013.

[47] N. Young. The k-server dual and loose competitiveness for paging. *Algorithmica*, 1994.

[48] Y. Zhou, J. Philbin, and K. Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *USENIX ATC*, 2001.