USENIX

Proceedings of FAST '12: 10th USENIX Conference on File and Storage Technologies *San Jose, CA, USA* *February 15–17, 2012*

Sponsored by
USENIX
in cooperation with
ACM SIGOPS

conference

proceedings

FAST '12: 10th USENIX Conference on File and Storage Technologies

San Jose, CA, USA
February 15–17, 2012

ISBN 978-1-931971-91-1

USENIX Association

Proceedings of FAST ’12: 10th USENIX Conference on File and Storage Technologies

February 15–17, 2012
San Jose, CA, USA

Conference Organizers

External Reviewers

FAST '12: 10th USENIX Conference on File and Storage Technologies
February 15–17, 2012
San Jose, CA, USA

Message from the Program Co-Chairs

Dear Colleagues,

We welcome you to the 10th USENIX Conference on File and Storage Technologies (FAST '12). This year we are proud to carry on the FAST tradition of presenting high-quality, innovative file and storage systems research. The program has a diverse set of papers on such topics as mobile and cloud storage systems, file system correctness, flash, deduplication, and the integration of new technologies such as GPUs. FAST continues to be a premier venue to bring together researchers and practitioners from the academic and industrial communities. This, too, is reflected in the program, which includes a balance of papers from universities and industry. Our community is increasingly an international one. This trend is reflected in the authors of this year's papers, whose affiliations come from nine countries and three continents.

FAST '12 received a record 137 submissions, of which 26 papers were selected, for an acceptance rate of 19%. Papers were reviewed using a two-round process. All papers received at least three reviews in the first round. Based on these reviews, 79 papers received two additional reviews. After vigorous electronic discussion, 56 of these papers were discussed at an all-day program committee meeting in Redmond, Washington. We used Eddie Kohler's excellent HotCRP software to handle paper submissions, reviews, PC discussion, and notifications.

This year, we introduced a new category of short papers to the program. Our intent was that such papers be held to the same high quality bar as regular submissions. Consequently, we did not separate such papers during the PC discussion, and we do not mark them separately in the program. We received 38 such submissions, of which 7 were accepted. The 18% acceptance rate was almost identical to the rate for full-length papers. We are hopeful that this format will prove to be a useful mechanism to report on results that may be inappropriate for a full-length submission.

We would like to thank everyone who contributed to this program. First and foremost, we are indebted to all the authors who submitted papers to FAST '12. We had a large body of high-quality work from which to select our program. We would also like to thank the attendees of FAST '12 and future readers of these papers. Together with the authors, you form the FAST community and make storage research vibrant and fun.

We would also like to recognize USENIX and the USENIX staff, who make all aspects of assembling a conference program easy. The USENIX staff provided outstanding support at all phases of this endeavor. They are largely responsible for the success of FAST this and every year. Thanks!

Finally, we would like to thank the Program Committee members for their countless hours and dedication. Due to the record number of papers we received, they had to put up with a heavier-than-anticipated review load. They, together with a few external reviewers, wrote 589 separate reviews comprising over 333,000 words of knowledgeable and insightful evaluation of the submitted work. In addition to writing reviews, the PC was very thoughtful during pre-meeting electronic discussions and at the PC meeting itself. We also thank our student scribe, Dutch Meyer, who took careful notes of the PC discussion so that we could report the content to authors.

We look forward to an interesting and enjoyable conference!

Bill Bolosky, Microsoft Research
Jason Flinn, University of Michigan
Program Co-Chairs

De-indirection for Flash-based SSDs with Nameless Writes

Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
Computer Sciences Department, University of Wisconsin-Madison

Abstract

We present *Nameless Writes*, a new device interface that removes the need for indirection in modern solid-state storage devices (SSDs). Nameless writes allow the device to choose the location of a write; only then is the client informed of the *name* (i.e., address) where the block now resides. Doing so allows the device to control block-allocation decisions, thus enabling it to execute critical tasks such as garbage collection and wear leveling, while removing the need for large and costly indirection tables. We demonstrate the effectiveness of nameless writes by porting the Linux ext3 file system to use an emulated nameless-writing device and show that doing so both reduces space and time overheads, thus making for simpler, less costly, and higher-performance SSD-based storage.

1 Introduction

Indirection is a core technique in computer systems [28]. Whether in the mapping of file names to blocks, or a virtual address space to an underlying physical one, system designers have applied indirection to improve system performance, reliability, and capacity for many years.

For example, modern hard disk drives use a modest amount of indirection to improve reliability by hiding underlying write failures. When a write to a particular physical block fails, a hard disk will *remap* the block to another location on the drive and record the mapping such that future reads will receive the correct data. In this manner, a drive transparently improves reliability without requiring any changes to the client above.

Indirection is particularly important in the new class of flash-based storage commonly referred to as Solid State Devices (SSDs). In modern SSDs, an indirection map in the Flash Translation Layer (FTL) enables the device to map writes in its virtual address space to any underlying physical location [11, 14, 16, 19, 21, 22].

FTLs use indirection for two reasons: first, to transform the erase/program cycle mandated by flash into the more typical write-based interface via copy-on-write techniques, and second, to implement *wear leveling* [18, 20], which is critical to increasing SSD lifetime. Because a flash block becomes unusable after a certain number of erase-program cycles (10,000 or 100,000 cycles according to manufacturers [8, 15]), such indirection is needed to spread the write load across flash blocks evenly and thus ensure that no particularly popular block causes the device to fail prematurely.

Unfortunately,the indirection such as found in many FTLs comes at a high price, which manifests as performance costs, space overheads, or both. If the FTL can flexibly map each virtual *page* in its address space (assuming a typical page size of 2 KB), an incredibly large indirection table is required. For example, a 1-TB SSD would need 2 GB of table space simply to keep one 32-bit pointer per 2-KB page of the device. Clearly, a completely flexible mapping is too costly; putting vast quantities of memory (usually SRAM) into an SSD is prohibitive.

Because of this high cost, most SSDs do not offer a fully flexible per-page mapping. A simple approach provides only a pointer per *block* of the SSD (a block typically contains 64 or 128 2-KB pages), which reduces overheads by the ratio of block size to page size. The 1-TB drive would now only need 32 MB of table space, which is more reasonable. However, as clearly articulated by Gupta et al. [16], block-level mappings have high performance costs due to excessive garbage collection.

As a result, the majority of FTLs today are built using a hybrid approach, mapping most data at block level and keeping a small page-mapped area for updates [11, 21, 22]. Hybrid approaches keep space overheads low while avoiding the high overheads of garbage collection, at the cost of additional device complexity. Unfortunately, garbage collection can still be costly, reducing the performance of the SSD, sometimes quite noticeably [16]. Regardless of the approach, FTL indirection incurs a significant cost; as SSDs scale, even hybrid schemes mostly based on block pointers will become infeasible.

In this paper, we introduce nameless writes, an approach that removes most of the costs of indirection in flash-based SSDs while still retaining its benefits. Our approach is a specific instance of *de-indirection*, in which an extra layer of indirection is removed. Unlike most writes, which specify both the *data* to write as well as a *name* (usually in the form of a logical address), a nameless write simply passes the data to the device. The device is free to choose any underlying physical block for the data; after the device *names* the block (i.e., decides where to write it), it informs the client of its choice. The client then can record the name for future reads.

One potential problem with nameless writes is the recursive update problem: if all writes are nameless, then any update to the file system requires a recursive set of updates up the file-system tree. To circumvent this problem, we introduce a *segmented address space*, which consists of a (large) physical address space for nameless writes, and a (small) virtual address space for traditional named writes. A file system running atop a nameless SSD can keep pointer-based structures in the virtual space; updates to those structures do not necessitate further updates up the tree, thus breaking the recursion.

Nameless writes offer great advantage over traditional writes, as they largely remove the need for indirection. Instead of pretending that the device can receive writes in any frequency to any block, a device that supports nameless writes is free to assign any physical page to a write when it is written; by returning the true name (i.e., the physical address) of the page to the client above (e.g., the file system), indirection is largely avoided, reducing the monetary cost of the SSD, improving its performance, and simplifying its internal structure.

Nameless writes (largely) remove the costs of indirection without giving away the primary responsibility an SSD manufacturer maintains: wear leveling. If an SSD simply exports the physical address space to clients, a simplistic file system or workload could cause the device to fail rather rapidly, simply by over-writing the same block repeatedly (whether by design or simply through a file-system bug). With nameless writes, no such failure mode exists. Because the device retains control of naming, it retains control of block placement, and thus can properly implement wear leveling to ensure a lengthy device lifetime. We believe that any solution that does not have this property is not viable, as no manufacturer would like to be so easily exposed to failure.

We demonstrate the benefits of nameless writes by porting the Linux ext3 file system to use a nameless SSD. Through extensive analysis on an emulated nameless SSD and comparison with different FTLs, we show the benefits of the new interface, in both reducing the space costs of indirection and improving random-write performance. Overall, we find that a nameless SSD uses a much smaller fraction of memory for indirection than a hybrid SSD while improving performance by an order of magnitude for some workloads.

The rest of this paper is organized as follows. In Section 2, we discuss the costs and benefits of indirection, and in Section 3 we present the nameless write interface. In Section 4, we show how to build a nameless-writing device. In Section 5, we describe how to port the Linux ext3 file system to use the nameless-writing interface, and in Section 6, we evaluate nameless writes through experimentation atop an emulated nameless-writing device. We discuss several related works in Section 7. Finally, in Section 8, we conclude and discuss our future work.

2 Indirection

It is said that "all problems in computer science can be solved by another level of indirection," a quote that is often attributed to Butler Lampson. Lampson, however, gives credit for this wisdom to David Wheeler, who not only uttered these famous words, but also usually added "...but that usually will create another problem [28]."

Indirection is a fundamental technique in computer systems. Before delving into the details of nameless writes, we first present a discussion of some of the general problems and solutions in systems that use indirection. First, we discuss why many systems utilize multiple levels of indirection, a problem we term *excess indirection*. We then describe the general solution to said problem, *de-indirection*, which removes an extra layer of indirection to improve performance or reduce space overheads.

2.1 Excess Indirection

Excess indirection exists in many systems that are widely used today, as well as in research prototypes. We now discuss four prominent examples: OS virtual memory running atop a hypervisor, a file system running atop a single disk, a file system atop a RAID array, and the focus of our work, file systems atop flash-based SSDs.

An excellent example of excess indirection arises in memory management of operating systems running atop hypervisors [9]. The OS manages virtual-to-physical mappings for each process that is running; the hypervisor, in turn, manages physical-to-machine mappings for each OS. In this manner, the hypervisor has full control over the memory of the system, whereas the OS above remains unchanged, blissfully unaware that it is not managing a real physical memory. Excess indirection leads to both space and time overheads in virtualized systems. The space overhead comes from maintaining OS physical addresses to machine addresses mapping for each page and from possible additional space overhead [1]. Time overheads exist as well in cases like the MIPS TLB-miss lookup in Disco [9].

Indirection also exists in modern disks. For example, modern disks maintain a small amount of extra indirection that maps bad sectors to nearby locations, in order to improve reliability in the face of write failures. Other examples include ideas for "smart" disks that remap writes in order to improve performance (for example, by writing to the nearest free location), which have been explored in previous research such as Loge [13] and "intelligent" disks [30]. These smart disks require large indirection tables inside the drive to map the logical address of the write to its current physical location. This requirement introduces new reliability challenges, including how to keep the indirection table persistent. Finally, fragmentation of randomly-updated files is also an issue.

File systems running atop modern RAID storage arrays provide another excellent example of excess indirection. Modern RAIDs often require indirection tables for fully-flexible control over the on-disk locations of blocks. In AutoRAID, a level of indirection allows the system to keep active blocks in mirrored storage for performance reasons, and move inactive blocks to RAID to increase effective capacity [32] and overcome the RAID small-update problem [26]. When a file system runs atop a RAID, excess indirection exists because the file system maps logical offsets to logical block addresses. The RAID, in turn, maps logical block addresses to physical (disk, offset) pairs. Such systems add memory space overhead to maintain these tables and meet the challenges of persisting the tables across power loss.

The focus of our work is flash-based SSDs, and thus it is no surprise that these too exhibit excess indirection. The extra level of indirection is provided via the Flash Translation Layer (FTL). The FTL is needed for two primary reasons. First, it is used to transform reads and writes issued by the client into reads and erase/program cycles supported by actual flash chips. In particular, because of the high cost of block erases (required before programming a page within the block), FTLs map current write activity to a small set of active blocks in a log-structured fashion, thus amortizing the cost of erases. Second, the FTL enables the SSD to implement wear leveling. Repeatedly erasing and programming a particular block will render it unreadable; thus, SSDs use the indirection provided by the FTL to spread write load across blocks and thus ensure that the device has a longer lifetime.

2.2 De-indirection

Because of these costs, system designers have long sought methods and techniques to reduce the costs of excess indirection in various systems. We label the removal of excess indirection *de-indirection*.

The basic idea is simple. Let us imagine a system with two levels of mapping, and thus excess indirection. The first indirection F maps items in the A space to items in the B space: $F(A_i) \rightarrow B_j$. The second indirection G maps items in the B space to those in the C space: $G(B_j) \rightarrow C_k$. To look up item i, one performs the following "excessive" indirection: $G(F(i))$.

De-indirection removes the second level of indirection by evaluating the second mapping $G()$ for all values mapped by $F()$: $\forall\, i : F(i) \leftarrow G(F(i))$. Thus, the top-level mapping simply extracts the needed values from the lower level indirection and installs them directly.

De-indirection has been successfully applied in a few domains, most notably within hypervisors. The Turtles project [7] provides an excellent example: in a recursively-virtualized environment (with hypervisors running on hypervisors), the Turtles system installs what the authors refer to as *multi-dimensional page tables*. Their approach essentially collapses multiple page tables into a single extra level of indirection, and thus reduces space and time overheads, making the costs of recursive virtualization more palatable.

2.3 Summary

Excess indirection is common across virtual memory and storage systems. In some cases, such as with hypervisor-based memory virtualization, it is required for functionality; each OS believes it owns the same physical memory, and thus cannot share it without the indirection provided by the hypervisor. In other cases, it improves performance, as we observed with disk systems and SSDs. Another reason for indirection is modularity and code simplicity. Finally, reliability is often the reason for excess indirection, notably within a single disk to handle write failures and within an SSD to perform wear leveling.

In all cases, at least part of the reason for excess indirection is the need to keep a fixed interface between higher and lower layers of the system. Without such a constraint, one could often remove the excess indirection and thus improve the system. For example, if an OS running on a para-virtualized system [31] is modified to request a machine page from the hypervisor and then install the correct virtual-to-machine page translation in its page tables, the hypervisor is relieved of having to manage this extra level of indirection, thus improving performance and reducing space overheads.

3 Nameless Writes

In this section, we discuss a new device interface that enables flash-based SSDs to remove a great deal of their infrastructure for indirection. We call a device that supports this interface a *Nameless-writing Device*. Table 1 summarizes the nameless-writing device interface.

The key feature of a nameless-writing device is its ability to perform nameless writes; however, to facilitate clients (such as file systems) to use a nameless-writing device, a number of other features are useful as well. In particular, the nameless-writing device should provide support for a segmented address space, migration callbacks, and associated metadata. We discuss these features in this section and how a prototypical file system could use them.

3.1 Nameless Write Interfaces

We first present the basic device interfaces of *Nameless Writes*: nameless (new) write, nameless overwrite, physical read, and free.

The nameless write interface completely replaces the existing write operation. A nameless write differs from a traditional write in two important ways. First, a nameless write does not specify a target address (i.e., a name); this allows the device to select the physical location without control from the client above. Second, after the device writes the data, it returns a *physical* address (i.e., a name)

Virtual Read	
down:	virtual address, length
up:	status, data
Virtual Write	
down:	virtual address, data, length
up:	status
Nameless Write	
down:	data, length, metadata
up:	status, resulting physical address(es)
Nameless Overwrite	
down:	old physical address(es), data, length, metadata
up:	status, resulting physical address(es)
Physical Read	
down:	physical address, length, metadata
up:	status, data
Free	
down:	virtual/physical addr, length, metadata, flag
up:	status
Migration [Callback]	
up:	old physical addr, new physical addr, metadata
down:	old physical addr, new physical addr, metadata

Table 1: **The Nameless-Writing Device Interfaces** *The table presents the nameless-writing device interfaces.*

and status to the client, which then keeps the name in its own structure for future reads.

The nameless overwrites interface is similar to the nameless (new) write interface, except that it also passes the old physical address(es) to the device. The device frees the data at the old physical address(es) and then performs a nameless write.

Read operations are mostly unchanged; as usual, they take as input the physical address to be read and return the data at that address and a status indicator. A slight change of the read interface is the addition of metadata in the input, for reasons that will be described in Section 3.4.

Because a nameless write is an allocating operation, a nameless-writing device needs to also be informed of de-allocation as well. Most SSDs refer to this interface as the *free* or *trim* command. Once a block has been freed (trimmed), the device is free to re-use it.

Finally, we consider how the nameless write interface could be utilized by a typical file-system client such as Linux ext3. For illustration, we examine the operations to append a new block to an existing file. First, the file system issues a nameless write of the newly-appended data block to a nameless-writing device. When the nameless write completes, the file system is informed of its address and can update the corresponding in-memory inode for this file so that it refers to the physical address of this block. Since the inode has been changed, the file system will eventually flush it to the disk as well; the inode must be written to the device with another nameless write. Again, the file system waits for the inode to be written and then updates any structures containing a reference to the inode. If nameless writes are the only interface available for writing to the storage device, then this recursion will continue until a root structure is reached. For file systems that do not perform this chain of updates or enforce such ordering, such as Linux ext2, additional ordering and writes are needed. This problem of recursive update has been solved in other systems by adding a level of indirection (e.g., the inode map in LFS [27]).

3.2 Segmented Address Space

To solve the recursive update problem without requiring substantial changes to the existing file system, we introduce a segmented address space with two segments (see Figure 1): the *virtual address space*, which uses virtual read, write and free interfaces, and the *physical address space*, which uses nameless read, write, overwrite, and free interfaces.

The virtual segment presents an address space from blocks 0 through $V - 1$, and is a virtual block space of size V blocks. The device virtualizes this address space, and thus keeps a (small) indirection table to map accesses to the virtual space to the correct underlying physical locations. Reads and writes to the virtual space are identical to reads and writes on typical devices. The client sends an address and a length (and, if a write, data) down to the device; the device replies with a status message (success or failure), and if a successful read, the requested data.

The nameless segment presents an address space from blocks 0 through $P - 1$, and is a physical block space of size P blocks. The bulk of the blocks in the device are found in this physical space, which allows typical named reads; however, all writes to physical space are nameless, thus preventing the client from directly writing to physical locations of its choice.

We use a virtual/physical flag to indicate the segment a block is in and the proper interface it should go through. The size of the two segments are not fixed. Allocation in either segment can be performed while there is still space on the device. A device space usage counter can be maintained for this purpose.

The reason for the segmented address space is to enable file systems to largely reduce the levels of recursive updates that would occur with only nameless writes. File systems such as ext2 and ext3 can be designed such that inodes and other metadata are placed in the virtual address space. Such file systems can simply issue a write to an inode and complete the update without needing to modify directory structures that reference the inode. Thus, the segmented address space allows updates to complete without propagating throughout the directory hierarchy.

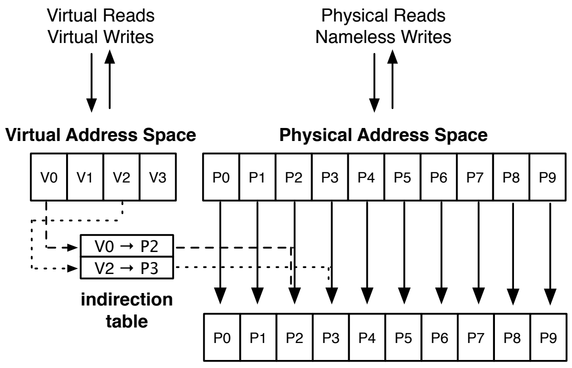


Figure 1: **The Segmented Address Space.** *A nameless-writing device provides a segmented address space to clients. The smaller virtual space allows normal reads and writes, which the device in turn maps to underlying physical locations. The larger physical space allows reads to physical addresses, but only nameless writes. In the example, only two blocks of the virtual space are currently mapped, V0 and V2, to physical blocks P2 and P3, respectively.*

3.3 Migration Callback

Several kinds of devices such as flash-based SSDs need to migrate data for reasons like wear leveling. We propose the *migration callback* interface to support such needs.

A typical flash-based SSD performs wear leveling via indirection: it simply moves the physical blocks and updates the map. With nameless writes, blocks in the physical segment cannot be moved without informing the file system. To allow the nameless-writing device to move data for wear leveling, a nameless-writing device uses *migration callbacks* to inform the file system of the physical address change of a block. The file system then updates any metadata pointing to this migrated block.

3.4 Associated Metadata

The final interface of a nameless-writing device is used to enable the client to quickly locate metadata structures that point to data blocks. The complete specification for associated metadata supports communicating metadata between the client and device. Specifically, the nameless write command is extended to include a third parameter: a small amount of metadata, which is persistently recorded adjacent to the data in a per-block header. Reads and migration callbacks are also extended to include this metadata. The associated metadata is kept with each block buffer in the page cache as well.

This metadata enables the client file system to readily identify the metadata structure(s) that points to a data block. For example, in ext3 we can locate the metadata structure that points to a data block by the inode number, the inode generation number, and the offset of the block in the inode. For file systems that already explicitly record back references, such as btrfs and NoFS [10], the back references can simply be reused for our purposes.

Such metadata structure identification can be used in several tasks. First, when searching for a data block in the page cache, we obtain the metadata information and compare it against the associated metadata of the data blocks in the page cache. Second, the migration callback process uses associated metadata to find the metadata that needs to be updated when a data block is migrated. Finally, associated metadata enables recovery in various crash scenarios, which we will discuss in detail in Section 5.7.

One last issue worth noticing is the difference between the associated metadata and address mapping tables. Unlike address mapping tables, the associated metadata is not used to locate physical data and is only used by the device during migration callbacks and crash recovery. Therefore, it can be stored adjacent to the data on the device. Only a small amount of the associated metadata is fetched into device cache for a short period of time during migration callbacks or recovery. Therefore, the space cost of associated metadata is much smaller than address mapping tables.

3.5 Implementation Issues

We now discuss various implementation issues that arise in the construction of a nameless-writing device. We focus on those issues different from a standard SSD, which are covered in detail elsewhere [16].

A number of issues revolve around the virtual segment. Most importantly, how big should such a segment be? Unfortunately, its size depends heavily on how the client uses it, as we will see when we port Linux ext3 to use nameless writes in Section 5. Our results in Section 6 show that a small virtual segment is usually sufficient.

The virtual space, by definition, requires an in-memory indirection table. Fortunately, this table is quite small, likely including simple page-level mappings for each page in the virtual segment. However, the virtual address space could be made larger than the size of the table; in this case, the device would have to swap pieces of the page table to and from the device, slowing down access to the virtual segment. Thus, while putting many data structures into the virtual space is possible, ideally the client should be miserly with the virtual segment, in order to avoid exceeding the supporting physical resources.

Another concern is the extra level of information naturally exported by exposing physical names to clients. Although the value of physical names has been extolled by others [12], a device manufacturer may feel that such information reveals too much of their "secret sauce" and thus be wary of adopting such an interface. We believe that if such a concern exists, the device could hand out modified forms of the true physical addresses, thus trying to hide the exact addresses from clients. Doing so may exact additional performance and space overheads, perhaps the cost of hiding information from clients.

4 Nameless-Writing Device

In this section, we describe our implementation of an emulated nameless-writing SSD. With nameless writes, a nameless-writing SSD can have a simpler FTL, which has the freedom to do its own allocation and wear leveling. We first discuss how we implement the nameless-writing interfaces and then propose a new garbage collection method that avoids file-system interaction. We defer the discussion of wear leveling to Section 5.6.

4.1 Nameless-Writing Interface Support

We implemented an emulated nameless-writing SSD that performs data allocation in a log-structured fashion by maintaining active blocks that are written in sequential order. When a nameless write is received, the device allocates the next free physical address, writes the data, and returns the physical address to the file system.

To support the virtual block space, the nameless-writing device maintains a mapping table between logical and physical addresses in its device cache. When the cache is full, the mapping table is swapped out to the flash storage of the SSD. As our results show in Section 6.1, the mapping table size of typical file system images is small; thus, such swapping rarely happens in practice.

The nameless-writing device handles trims in a manner similar to traditional SSDs; it invalidates the physical address sent by a trim command. During garbage collection, invalidated pages can be recycled. The device also invalidates the old physical addresses of overwrites.

A nameless-writing device needs to keep certain associated metadata for nameless writes. We choose to store the associated metadata of a data page in its Out-Of-Band (OOB) area. The associated metadata is moved together with data pages when the device performs a migration.

4.2 In-place Garbage Collection

In this section, we describe a new garbage collection method for nameless-writing devices. Traditional FTLs perform garbage collection on a flash block by reclaiming its invalid data pages and migrating its live data pages to new locations. Such garbage collection requires a nameless-writing device to inform the file system of the new physical addresses of the migrated live data; the file system then needs to update and write out its metadata. To avoid the costs of such callbacks and additional metadata writes, we propose *in-place garbage collection*, which writes the live data back to the same location instead of migrating it. A similar hole-plugging approach was proposed in earlier work [24], where live data is used to plug the holes of most utilized segments.

To perform in-place garbage collection, the FTL selects a candidate block using a certain policy. The FTL reads all live pages from the chosen block together with their associated metadata, stores them temporarily in a super-capacitor- or battery-backed cache, and then erases the block. The FTL next writes the live pages to their original addresses and tries to fill the rest of the block with writes in the waiting queue of the device. Since a flash block can only be written in one direction, when there are no waiting writes to fill the block, the FTL marks the free space in the block as unusable. We call such space *wasted space*. During in-place garbage collection, the physical addresses of live data are not changed. Thus, no file system involvement is needed.

Policy to choose candidate block: A natural question is how to choose blocks for garbage collection. A simple method is to pick blocks with the fewest live pages so that the cost of reading and writing them back is minimized. However, choosing such blocks may result in an excess of wasted space. In order to pick a good candidate block for in-place garbage collection, we aim to minimize the cost of rewriting live data and to reduce wasted space during garbage collection. We propose an algorithm that tries to maximize the benefit and minimize the cost of in-place garbage collection. We define the cost of garbage collecting a block to be the total cost of erasing the block (T_{erase}), reading (T_{page_read}) and writing (T_{page_write}) live data (N_{valid}) in the block.

$$cost = T_{erase} + (T_{page\_read} + T_{page\_write}) * N_{valid}$$

We define benefit as the number of new pages that can potentially be written in the block. Benefit includes the following items: the current number of waiting writes in the device queue (N_{wait_write}), which can be filled into empty pages immediately, the number of empty pages at the end of a block (N_{last}), which can be filled at a later time, and an estimated number of future writes based on the speed of incoming writes (S_{write}). While writing valid pages (N_{valid}) and waiting writes (N_{wait_write}), new writes will be accumulated in the device queue. We account for these new incoming writes by $T_{page\_write} * (N_{valid} + N_{wait\_write}) * S_{write}$. Since we can never write more than the amount of the recycled space (i.e., number of invalid pages, $N_{invalid}$) of a block, the benefit function uses the minimum of the number of invalid pages and the number of all potential new writes.

$$\begin{aligned} benefit = \min(&N_{invalid}, N_{wait\_write} + N_{last} \\ &+ T_{page\_write} * (N_{valid} + N_{wait\_write}) * S_{write}) \end{aligned}$$

The FTL calculates the $\frac{benefit}{cost}$ ratio of all blocks that contain invalid pages and selects the block with the maximal ratio to be the garbage collection candidate. Computationally less expensive algorithms could be used to find reasonable approximations; such an improvement is left to future work.

5 Nameless Writes on ext3

In this section we discuss our implementation of nameless writes on the Linux ext3 file system. The Linux ext3 file system is a classic journaling file system that is commonly used in many Linux distributions. It extends the Linux ext2 file system and uses the same allocation method as ext2. It provides three journaling modes: data mode, ordered mode, and journal mode. The ordered journaling mode of ext3 is a commonly used mode, which writes metadata to the journal and writes data to disk before committing metadata of the transaction. It provides ordering that can be naturally used by nameless writes, since the nameless-writing interface requires metadata to reflect physical address returned by data writes. When committing metadata in ordered mode, the physical addresses of data blocks are known to the file system because data blocks are written out first. Thus, we implemented nameless writes with ext3 ordered mode; other modes are left for future work.

5.1 Segmented Address Space

We first discuss physical and virtual address space separation and modified file-system allocation on ext3. We use the physical address space to store all data blocks and the virtual address space to store all metadata structures, including superblocks, inodes, data and inode bitmaps, indirect blocks, directory blocks, and journal blocks. We use the type of a block to determine whether it is in the virtual or the physical address space and the type of interface it goes through.

The nameless-writing file system does not perform allocation of the physical address space and only allocates metadata in the virtual address space. Therefore, we do not fetch or update group bitmaps for nameless block allocation. For these data blocks, the only bookkeeping task that the file system needs to perform is tracking overall device space usage. Specifically, the file system checks for total free space of the device and updates the free space counter when a data block is allocated or de-allocated. Metadata blocks in the virtual physical address space are allocated in the same way as the original ext3 file system, thus making use of existing bitmaps.

5.2 Associated Metadata

We include the following items as associated metadata of a data block: 1) the inode number or the logical address of the indirect block that points to the data block, 2) the offset within the inode or the indirect block, 3) the inode generation number, and 4) a timestamp of when the data block was last updated. Items 1 to 3 are used to identify the metadata structure that points to a data block. Item 4 is used during the migration callback process to update the metadata structure with the most up-to-date physical address of a data block.

All the associated metadata is stored in the OOB area of a data block. The total amount of additional status we store in the OOB area is less than 48 bytes, smaller than the typical 128-byte OOB size of 4-KB flash pages. For reliability reasons, we assume that a data page and its OOB area are always written atomically.

5.3 Write

To perform a nameless write, the file system sends the data and the associated metadata of the block to the device. When the device finishes a nameless write and sends back its physical address, the file system updates the inode or the indirect block pointing to it with the new physical address. It also updates the block buffer with the new physical address. In ordered journaling mode, metadata blocks are always written after data blocks have been committed; thus on-disk metadata is always consistent with its data. The file system performs overwrites similarly. The only difference is that overwrites have an existing physical address, which is sent to the device; the device uses this information to invalidate the old data.

5.4 Read

We change two parts of the read operation of data blocks in the physical address space: reading from the page cache and reading from the physical device. To search for a data block in the page cache, we compare the metadata index (e.g., inode number, inode generation number, and block offset) of the block to be read against the metadata associated with the blocks in the page cache. If the buffer is not in the page cache, the file system fetches it from the device using its physical address. The associated metadata of the data block is also sent with the read operation to enable the device to search for remapping entries during device wear leveling (see Section 5.6).

5.5 Free

The current Linux ext3 file system does not support the SSD trim operation. We implemented the ext3 trim operation in a manner similar to ext4. Trim entries are created when the file system deletes a block (named or nameless). A trim entry contains the logical address of a named block or the physical address of a nameless block, the length of the block, its associated metadata, and the address space flag. The file system then adds the trim entry to the current journal transaction. At the end of transaction commit, all trim entries belonging to the transaction are sent to the device. The device locates the block to be deleted using the information contained in the trim operation and invalidates the block.

When a metadata block is deleted, the original ext3 de-allocation process is performed. When a data block is deleted, no de-allocation is performed (i.e., bitmaps are not updated); only the free space counter is updated.

5.6 Wear Leveling with Callbacks

When a nameless-writing device performs wear leveling, it migrates live data to achieve even wear of the device. When such migration happens with data blocks in the physical address space, the file system needs to be informed about the change of their physical addresses. In this section, we describe how the nameless-writing device handles data block migration and how it interacts with the file system to perform *migration callbacks*.

When live nameless data blocks (together with their associated metadata in the OOB area) are migrated during wear leveling, the nameless-writing device creates a mapping from the data block's old physical address to its new physical address and stores it together with its associated metadata in a *migration mapping table* in the device cache. The migration mapping table is used to locate the migrated physical address of a data block for reads and overwrites, which may be sent to the device with the block's old physical address. After the mapping has been added, the old physical address is reclaimed and can be used by future writes.

At the end of a wear-leveling operation, the device sends a migration callback to the file system, which contains all migrated physical addresses and their associated metadata. The file system then uses the associated metadata to locate the metadata pointing to the data block and updates it with the new physical address in a background process. Next, the file system writes changed metadata to the device. When a metadata write finishes, the file system deletes all the callback entries belonging to this metadata block and sends a response to the device, informing it that the migration callback has been processed. Finally, the device deletes the remapping entry when receiving the response of a migration callback.

For migrated metadata blocks, the file system does not need to be informed of the physical address change since it is kept in the virtual address space. Thus, the device does not keep remapping entries or send migration callbacks for metadata blocks.

During the migration callback process, we allow reads and overwrites to the migrated data blocks. When receiving a read or an overwrite during the callback period, the device first looks in the migration mapping table to locate the current physical address of the data block and then performs the request.

Since all remapping entries are stored in the on-device RAM before the file system finishes processing the migration callbacks, we may run out of RAM space if the file system does not respond to callbacks or responds too slowly. In such a case, we simply prohibit future wear-leveling migrations and prevent block wear-out only through garbage collection.

5.7 Reliability Discussion

The changes of the ext3 file system discussed above may cause new reliability issues. In this section, we discuss several reliability issues and our solutions to them.

There are three main reliability issues related to nameless writes. First, we maintain a mapping table in the on-device RAM for the virtual address space. This table needs to be reconstructed each time the device powers on (either after a normal power-off or a crash). Second, the in-memory metadata can be inconsistent with the physical addresses of nameless blocks because of a crash after writing a data block and before updating its metadata block, or because of a crash during wear-leveling callbacks. Finally, crashes can happen during in-place garbage collection, specifically, after reading the live data and before writing them back, which may cause data loss.

We solve the first two problems by using the metadata information maintained in the device OOB area. We store logical addresses with data pages in the virtual address space for reconstructing the logical-to-physical address mapping table. We store associated metadata, as discussed in Section 3.4, with all nameless data. We also store the validity of all flash pages in their OOB area. We maintain an invariant that metadata in the OOB area is always consistent with the data in the flash page by writing the OOB area and the flash page atomically.

We solve the in-place garbage collection reliability problem by requiring the use of a small memory backed by battery or super-capacitor. Notice that the amount of live data we need to hold during a garbage collection operation is no more than the size of an SSD block, typically 256 KB, thus only adding a small monetary cost to the whole device.

The recovery process works as follows. When the device is started, we perform a whole-device scan and read the OOB area of all valid flash pages to reconstruct the mapping table of the virtual address space. If a crash is detected, we perform the following steps. The device sends the associated metadata in the OOB area and the physical addresses of flash pages in the physical address space to the file system. The file system then locates the proper metadata structures. If the physical address in a metadata structure is inconsistent, the file system updates it with the new physical address and adds the metadata write to a dedicated transaction. After all metadata is processed, the file system commits the transaction, at which point the recovery process is finished.

6 Evaluation

In this section, we present our evaluation of nameless writes on an emulated nameless-writing device. Specifically, we focus on studying the following questions:

- What are the space costs of nameless-writing devices compared to other FTLs?

Configuration	Value
SSD Size	4 GB
Page Size	4 KB
Block Size	256 KB
Number of Planes	10
Hybrid Log Block Area	5%
Page Read Latency	25 μs
Page Write Latency	200 μs
Block Erase Latency	1500 μs

Table 2: **SSD Emulator Configuration.**

Image Size	Page	Hybrid	Nameless
328 MB	328 KB	38 KB	2.7 KB
2 GB	2 MB	235 KB	12 KB
10 GB	10 MB	1.1 MB	31 KB
100 GB	100 MB	11 MB	251 KB
400 GB	400 MB	46 MB	1 MB
1 TB	1 GB	118 MB	2.2 MB

Table 3: **FTL Mapping Table Size.** *Mapping table size of page-level, hybrid, and nameless-writing devices with different file system images. The configuration in Table 2 is used.*

- What is the overall performance benefit of nameless-writing devices?
- What is the write performance of nameless-writing devices? How and why is it different from page-level mapping and hybrid mapping FTLs?
- What is the cost of in-place garbage collection and the overhead of wear-leveling callbacks?
- Is crash recovery correct and what are its overheads?

SSD Emulator: We built an SSD emulator which models a multi-plane SSD with garbage collection and wear leveling as a pseudo block device based on David [4]. We implemented three types of FTLs: page-level mapping, hybrid mapping and nameless-writing on top of the PSU objected-oriented SSD simulator codebase [6]. Data is stored in memory to enable quick and accurate emulation. Table 2 describes the configuration we used.

The page-level mapping FTL writes data in a log-structured fashion and schedules in round-robin order across parallel planes. It keeps a mapping for each data page between its logical and physical address. We assume (unrealistically) that this SSD has enough memory to store all page-level mappings. The page-level SSD serves as an upper-bound on performance.

We implemented a hybrid mapping FTL similar to FAST [22], which uses a *log block area* for random data and one sequential log block dedicated for sequential streams. The rest of the device is a *data block area* used to store whole data blocks. The hybrid mapping FTL maintains the page-level mapping of the log block area and the block-level mapping of the data block area.

We implemented a simple garbage collection algorithm that recycles blocks with the least live data in page-level mapping and hybrid mapping FTLs, and a wear-leveling algorithm on all three FTLs that considers a block's remaining erase cycles and its data temperature during wear leveling similar to a previous wear-leveling algorithm [2].

System Setup: We implemented the emulated nameless-write device and the nameless-writing ext3 file system on a 64-bit Linux 2.6.33 kernel. The page-level mapping and the hybrid mapping SSD emulators are built on an unmodified 64-bit Linux 2.6.33 kernel. All experiments are performed on a 2.5 GHz Intel Quad Core CPU with 8 GB memory.

6.1 SSD Memory Consumption

We first study the space cost of mapping tables used by different SSD FTLs: nameless-writing, page-level mapping, and hybrid mapping. The mapping table size of page-level and hybrid FTLs is calculated based on the total size of the device, its block size, and its log block area size (for hybrid mapping). A nameless-writing device keeps a mapping table for the entire file system's virtual address space. Since we map all metadata to the virtual block space in our nameless-writing implementation, the mapping table size of the nameless-writing device is dependent on the metadata size of the file system image. We use Impressions [3] to create typical file system images of sizes up to 1 TB and calculate their metadata sizes.

Figure 3 shows the mapping table sizes of the three FTLs with different file system images produced by Impressions. Unsurprisingly, the page-level mapping has the highest mapping table space cost. The hybrid mapping has a moderate space cost; however, its mapping table size is still quite large: over 100 MB for a 1-TB device. The nameless mapping table has the lowest space cost; even for a 1-TB device, its mapping table uses less than 3 MB of space for typical file systems, reducing both cost and power usage.

6.2 Application Performance

We now present the overall application performance of nameless-writing, page-level mapping and hybrid mapping FTLs with macro-benchmarks. We use varmail, fileserver, and webserver from the filebench suite [29].

Figure 2 shows the throughput of these benchmarks. We see that both page-level mapping and nameless-writing FTLs perform better than the hybrid mapping FTL with varmail and fileserver. These benchmarks contain 90.8% and 70.6% random writes, respectively. As we will see later in this section, the hybrid mapping FTL performs well with sequential writes and poorly with random writes. Thus, its throughput for these two benchmarks

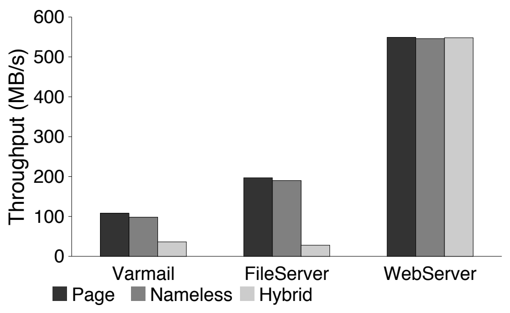


Figure 2: **Throughput of Filebench.** *Throughput of varmail, fileserver, and webmail macro-benchmarks with page-level, nameless-writing, and hybrid FTLs.*

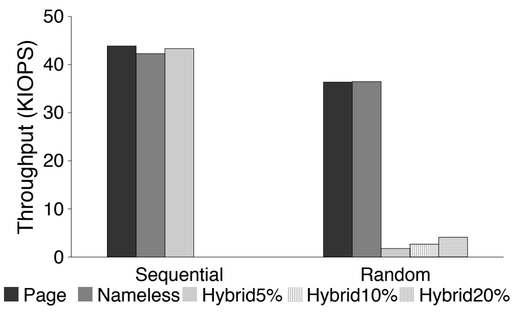


Figure 3: **Sequential and Random Write Throughput.** *Throughput of sequential writes and sustained 4-KB random writes. Random writes are performed over a 2-GB range.*

is worse than the other two FTLs. For webserver, all three FTLs deliver similar performance, since it contains only 3.8% random writes. We see a small overhead of the nameless-writing FTL as compared to the page-level mapping FTL with all benchmarks, which we will discuss in detail in Sections 6.5 and 6.6.

In summary, we demonstrate that the nameless-writing device achieves excellent performance, roughly on par with the costly page-level approach, which serves as an upper-bound on performance.

6.3 Basic Write Performance

Write performance of flash-based SSDs is known to be much worse than read performance, with random writes being the performance bottleneck. Nameless writes aim to improve write performance of such devices by giving the device more data-placement freedom. We evaluate the basic write performance of our emulated nameless-writing device in this section. Figure 3 shows the throughput of sequential writes and sustained 4-KB random writes with page-level mapping, hybrid mapping, and nameless-writing FTLs.

First, we find that the emulated hybrid-mapping device has a sequential throughput of 169 MB/s and a sustained 4-KB random write throughput of 2,830 IOPS. A widely used real middle-end SSD has sequential throughput of up to 70 MB/s and random throughput of up to 3,300 IOPS [17]. Although the write performance of our emulator does not match this real SSD exactly, it is still in the ballpark of actual SSD performance, and thus useful in our study. The goal of our hybrid-mapping emulator is not to model one particular SSD perfectly but to provide insight into the fundamental problems of hybrid-mapped SSDs as compared to page-mapped and nameless SSDs.

Second, the random write throughput of page-level mapping and nameless-writing FTLs is close to their sequential write throughput, because both FTLs allocate data in a log-structured fashion, making random writes behave like sequential writes. The overhead of random writes with these two FTLs comes from their garbage collection process. Since whole blocks can be erased when they are overwritten in sequential order, garbage collection has the lowest cost with sequential writes. By contrast, garbage collection of random data may incur the cost of live data migration.

Third, we notice that the random write throughput of the hybrid mapping FTL is significantly lower than that of the other FTLs and its own sequential write throughput. The poor random write performance of the hybrid mapping FTL results from the costly full-merge operation and its corresponding garbage collection process [16]. Full merges are required each time a log block is filled with random writes, thus a dominating cost for random writes.

One way to improve the random write performance of hybrid-mapped SSDs is to over-provision more log block space. To explore that, we vary the size of the log block area with the hybrid mapping FTL from 5% to 20% of the whole device and found that random write throughput gets higher as the size of the log block area increases. However, only the data block area reflects the effective size of the device, while the log block area is part of device over-provisioning. Therefore, hybrid-mapped SSDs often sacrifice device space cost for better random write performance. Moreover, the hybrid mapping table size increases with higher log block space, requiring larger on-device RAM. Nameless writes achieve significantly better random write performance with no additional over-provisioning or RAM space.

Finally, Figure 3 shows that the nameless-writing FTL has low overhead as compared to the page-level mapping FTL with sequential and random writes. We explain this result in more detail in Section 6.5 and 6.6.

6.4 A Closer Look at Random Writes

A previous study [16] and our study in the last section show that random writes are the major performance bottleneck of flash-based devices. We now study two subtle yet fundamental questions: do nameless-writing devices

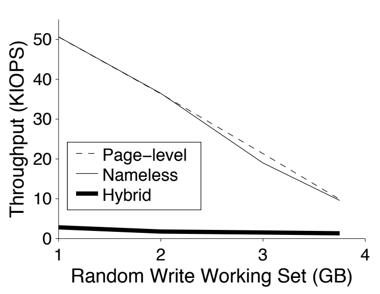


Figure 4: **Random Write Throughput.** *Throughput of sustained 4-KB random writes over different working set sizes with page-level, nameless, and hybrid FTLs.*

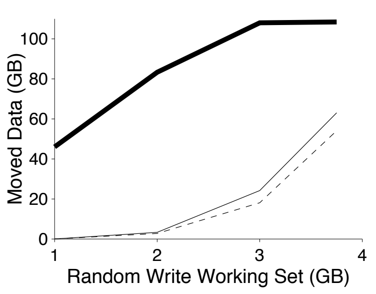


Figure 5: **Migrated Live Data.** *Amount of migrated live data during garbage collection of random writes with different working set sizes with page-level, nameless, and hybrid FTLs.*

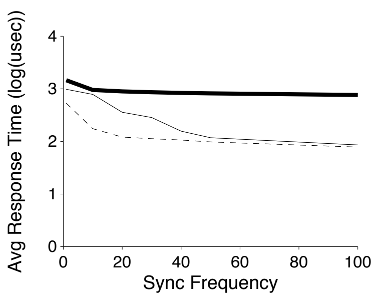


Figure 6: **Average Response Time of Synchronous Random Writes.** *4-KB random writes in a 2-GB file. Sync frequency represents the number of writes we issue before calling an fsync.*

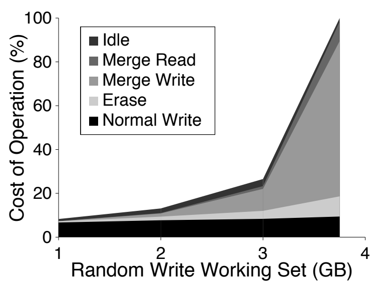


Figure 7: **Page-Level FTL Utilization.** *Break down of device utilization with the page-level FTL under random writes of different ranges.*

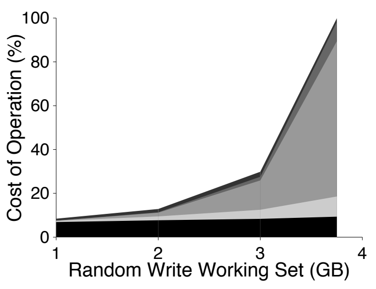


Figure 8: **Nameless FTL Utilization.** *Break down of device utilization with the nameless FTL under random writes of different ranges.*

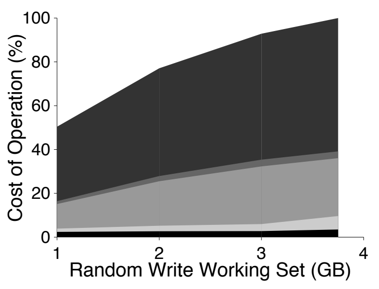


Figure 9: **Hybrid FTL Utilization.** *Break down of device utilization with the hybrid FTL under random writes of different ranges.*

perform well with different kinds of random-write workloads, and why do they outperform hybrid devices.

To answer the first question, we study the effect of working set size on random writes. We create files of different sizes and perform sustained 4-KB random writes in each file to model different working set sizes. Figure 4 shows the throughput of random writes over different file sizes with all three FTLs. We find that the working set size has a large effect on random write performance of nameless-writing and page-level mapping FTLs. The random write throughput of these FTLs drops as the working set size increases. When random writes are performed over a small working set, they will be overwritten in full when the device fills and garbage collection is triggered. In such cases, there is a higher chance of finding blocks that are filled with invalid data and can be erased with no need to rewrite live data, thus lowering the cost of garbage collection. In contrast, when random writes are performed over a large working set, garbage collection has a higher cost since blocks contain more live data, which must be rewritten before erasing a block.

To further understand the increasing cost of random writes as the working set increases, we plot the total amount of live data migrated during garbage collection (Figure 5) of random writes over different working set sizes with all three FTLs. This graph shows that as the working set size of random writes increases, more live data is migrated during garbage collection for these FTLs, resulting in a higher garbage collection cost and worse random write performance.

Comparing the page-level mapping FTL and the nameless-writing FTL, we find that nameless-writing has slightly higher overhead when the working set size is high. This overhead is due to the cost of in-place garbage collection when there is wasted space in the recycled block. We will study this overhead in details in the next section.

We now study the second question to further understand the cost of random writes with different FTLs. We break down the device utilization into regular writes, block erases, writes during merging, reads during merging, and device idle time. Figures 7, 8, and 9 show the stack plot of these costs over all three FTLs. For page-level mapping and nameless-writing FTLs, we see that the major cost comes from regular writes when random writes are performed over a small working set. When the working set increases, the cost of merge writes and erases increases

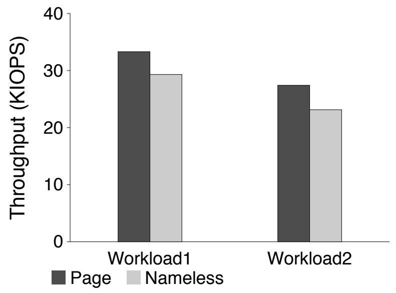


Figure 10: **Write Throughput with Wear leveling.** *Throughput of biased sequential writes with wear leveling under page-level and nameless FTLs.*

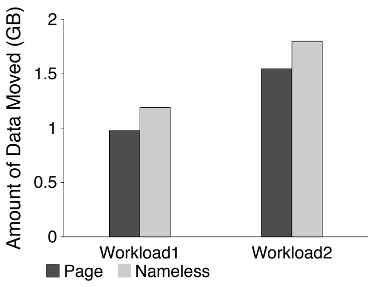


Figure 11: **Migrated Live Data during Wear Leveling.** *Amount of migrated live data during wear leveling under page-level and nameless FTLs.*

	Metadata	RemapTbl
Workload1	2.02 MB	321 KB
Workload2	5.09 MB	322 KB

Figure 12: **Wear leveling Callback Overhead.** *Amount of additional metadata writes because of migration callbacks and maximal remapping table size during wear leveling with the nameless-writing FTL.*

and becomes the major cost. For the hybrid mapping FTL, the major cost of random writes comes from migrating live data and idle time during merging for all working set sizes. When the hybrid mapping FTL performs a full merge, it reads and writes pages from different planes, thus creating idle time on each plane.

In summary, we demonstrate that random write throughput of the nameless-writing FTL is close to that of the page-level mapping FTL and is significantly better than the hybrid mapping FTL, mainly because of the costly merges the hybrid mapping FTL performs for random writes. We also found that both nameless-writing and page-level mapping FTLs achieve better random write throughput when the working set is relatively small because of a lower garbage collection cost.

6.5 In-place Garbage Collection Overhead

The performance overhead of a nameless-writing device may come from two different device responsibilities: garbage collection and wear leveling. We study the overhead of in-place garbage collection in this section and wear-leveling overhead in the next section.

Our implementation of the nameless-writing device uses an in-place merge to perform garbage collection. As explained in Section 4.2, when there are no waiting writes on the device, we may waste the space that has been recently garbage collected. We use synchronous random writes to study this overhead. We vary the frequency of calling *fsync* to control the amount of waiting writes on the device; when the sync frequency is high, there are fewer waiting writes on the device queue. Figure 6 shows the average response time of 4-KB random writes with different sync frequencies under page-level mapping, nameless-writing, and hybrid mapping FTLs. We find that when sync frequency is high, the nameless-writing device has a larger overhead compared to page-level mapping. This overhead is due to the lack of waiting writes on the device to fill garbage-collected space. However, we see that the average response time of the nameless-writing FTL is still lower than that of the hybrid mapping FTL, since response time is worse when the hybrid FTL performs full-merge with synchronous random writes.

6.6 Wear-leveling Callback Overhead

Finally, we study the overhead of wear leveling in a nameless-writing device. To perform wear-leveling experiments, we reduce the lifetime of SSD blocks to 50 erase cycles. We set the threshold of triggering wear leveling to be 75% of the maximal block lifetime, and set blocks that are under 90% of the average block remaining time to be candidates for wear leveling.

We create two workloads to model different data temperature and SSD wear: a workload that first writes 3.5-GB data in sequential order and then overwrites the first 500-MB area 40 times (Workload 1), and a workload that overwrites the first 1-GB area 40 times (Workload 2). Workload 2 has more hot data and triggers more wear leveling. We compare the throughput of these workloads with page-level mapping and nameless-writing FTLs in Figure 10. The throughput of Workload 2 is worse than that of Workload 1 because of its more frequent wear-leveling operation. Nonetheless, the performance of the nameless-writing FTL with both workloads has less than 9% overhead.

We then plot the amount of migrated live data during wear leveling with both FTLs in Figure 11. As expected, Workload 2 produces more wear-leveling migration traffic. Comparing page-level mapping to nameless-writing FTLs, we find that the nameless-writing FTL migrates more live data. When the nameless-writing FTL performs in-place garbage collection, it generates more migrated live data, as shown in Figure 5. Therefore, more erases are caused by garbage collection with the nameless-writing FTL, resulting in more wear-leveling invocation and more wear-leveling migration traffic.

Migrating live nameless data in a nameless-writing device creates callback traffic and additional metadata writes. Wear leveling in a nameless-writing device also adds a space overhead when it stores the remapping table for migrated data. We show the amount of additional metadata writes and the maximal size of the remapping table of a nameless-writing device in Figure 12. We find both overheads to be low with the nameless-writing device: an addition of less than 6 MB metadata writes and a space cost of less than 350 KB.

In summary, we find that both the garbage-collection and wear-leveling overheads caused by nameless writes are low. Since wear leveling is not a frequent operation and is often scheduled in system idle periods, we expect both performance and space overheads of a nameless-writing device to be even lower in real systems.

6.7 Reliability

To determine the correctness of our reliability solution, we inject crashes in the following points: 1) after writing a data block and its metadata block, 2) after writing a data block and before updating its metadata block, 3) after writing a data block and updating its metadata block but before committing the metadata block, and 4) after the device migrates a data block because of wear leveling and before the file system processes the migration callback. In all cases, we successfully recover the system to a consistent state that correctly reflects all written data blocks and their metadata.

Our results also show that the overhead of our crash recovery process is relatively small: from 0.4 to 6 seconds, depending on the amount of inconsistent metadata after crash. With more inconsistent metadata, the overhead of recovery is higher.

7 Related Work

A large body of work on flash-based SSD FTLs and file systems that manage them has been proposed in recent years [11, 14, 16, 19, 21, 22, 25, 33]. In this section, we discuss the two research projects that are most related to nameless writes.

Range writes [5] use an approach similar to nameless writes. Range writes were proposed to improve hard disk performance by letting the file system specify a range of addresses and letting the device pick the final physical address of a write. Instead of a range of addresses, nameless writes are not specified with any addresses, thus obviating file system allocation and moving allocation responsibility to the device. Problems such as updating metadata after writes in range writes also arise in nameless writes. We propose a segmented address space to lessen the overhead and the complexity of such an update process. Another difference is that nameless writes target devices that need to maintain control of data placement, such as wear leveling in flash-based devices. Range writes target traditional hard disks that do not have such responsibilities. Data placement with flash-based devices is also less restricted than traditional hard disks, since flash-based memory has uniform access latency regardless of its location.

The poor random write performance of hybrid FTLs has drawn attention from researchers in recent years. The demand-based Flash Translation Layer (DFTL) was proposed to address this problem by maintaining a page-level mapping table and writing data in a log-structured fashion [16]. DFTL stores its page-level mapping table on the device and keeps a small portion of the mapping table in the device cache based on workload temporal locality. However, for workloads that have a bigger working set than the device cache, swapping the cached mapping table with the on-device mapping table structure can be costly. There is also a space overhead to store the entire page-level mapping table on device. We use a log-structured write order similar to DFTL to maximize the device's sequential writing capability. However, the need for a device-level mapping table is obviated with nameless writes. Indirection is maintained only for the virtual address space, which as we show, requires a small space cost and can fit in the device cache with typical file system images. Thus, we do not pay the space cost of storing the large page-level mapping table in the device or the performance overhead of swapping mapping table entries.

8 Conclusions and Future Work

In this paper, we introduced nameless writes, a new write interface built to reduce the inherent costs of indirection. Through the implementation of nameless writes on the Linux ext3 file system and an emulated nameless-writing device, we demonstrate how to port a file system to use nameless writes. Through extensive evaluations, we show the great advantage of nameless writes: greatly reduced space costs and improved random-write performance.

Porting other types of file systems to use nameless writes would be interesting and is a part of our future work. Here, we give a brief discussion about these file systems and the challenges we foresee in changing them to use nameless writes.

Linux ext2: The Linux ext2 file system is similar to the ext3 file system except that it has no journaling. While we rely on the ordered journal mode to provide a natural ordering for the metadata update process of nameless writes in ext3, we need to introduce an ordering on the ext2 file system. Our initial implementation of nameless-writing ext2 shows that one possible method to enforce such an ordering is to defer metadata writes until all the ongoing data writes belonging to them have finished.

Copy-On-Write File Systems and Snapshots: As an alternative to journaling, *copy-on-write* (COW) file systems always write out updates to new free space; when all of those updates have reached the disk, a root structure is updated to point at the new structures, and thus include them in the state of the file system. COW file systems thus map naturally to nameless writes. All writes to free space are mapped into the physical segment and issued namelessly; the root structure is mapped into the virtual segment. The write ordering is not affected, as COW file systems all must wait for the COW writes to complete before issuing a write to the root structure anyway.

One problem with COW file systems or other file systems that support snapshots or versions is that multiple metadata structures can point to the same data block, which may result in a large amount of associated metadata. We can use file system intrinsic back references, such as those in btrfs, or structures like *Backlog* [23] to represent associated metadata. Another problem is that multiple metadata blocks need to be updated after a nameless write. One possible way to control the number of metadata updates is to reduce the amount of metadata included in the virtual address space.

Extent-Based File Systems: One final type of file systems worth considering are *extent-based* file systems, such as Linux btrfs and ext4, where contiguous regions of a file are pointed to via (pointer, length) pairs instead of a single pointer per fixed-sized block. Modifying an extent-based file system to use nameless writes would require a bit of work; as nameless writes of data are issued, the file system would not (yet) know if the data blocks will form one extent or many. Thus, only when the writes complete will the file system be able to determine the outcome. Later writes would not likely be located nearby, and thus to minimize the number of extents, updates should be issued at a single time. Extents also hint at the possibility of a new interface for nameless writes. Specifically it might be useful to provide an interface to *reserve* a larger contiguous region on the device; doing so would enable the file system to ensure that a large file was placed contiguously in physical space, and thus affords a highly compact extent-based representation. We plan to look into such enhancements in the future.

Acknowledgment

We thank the anonymous reviewers and Jason Flinn (our shepherd) for their tremendous feedback and comments, which have substantially improved the content and presentation of this paper. We also thank the members of the ADSL research group for their insightful comments.

This material is based upon work supported by the National Science Foundation under the following grant: NSF CCF-0937959, as well as by generous donations from Google, NetApp, and Samsung.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

References

[1] K. Adams and O. 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] N. Agarwal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX Annual Technical Conference (USENIX '08)*, Boston, Massachusetts, June 2008.

[3] N. Agrawal, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Generating Realistic Impressions for File-System Benchmarking. In *Proceedings of the 7th USENIX Symposium on File and Storage Technologies (FAST '09)*, San Francisco, California, February 2009.

[4] N. Agrawal, L. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Emulating Goliath Storage Systems with David. In *Proceedings of the 9th USENIX Symposium on File and Storage Technologies (FAST '11)*, San Jose, California, February 2011.

[5] A. Anand, S. Sen, A. Krioukov, F. Popovici, A. Akella, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Banerjee. Avoiding File System Micromanagement with Range Writes. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, California, December 2008.

[6] B. Tauras, Y. Kim, and A. Gupta. PSU Objected-Oriented Flash based SSD simulator. http://csl.cse.psu.edu/?q=node/321.

[7] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, Canada, December 2010.

[8] S. Boboila and P. Desnoyers. Write Endurance in Flash Drives: Measurements and Analysis. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST '10)*, San Jose, California, February 2010.

[9] E. Bugnion, S. Devine, and M. 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.

[10] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Consistency Without Ordering. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, San Jose, California, February 2012.

[11] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song. System Software for Flash Memory: A Survey. In *Proceedings of thei 5th International Conference on Embedded and Ubiquitous Computing (EUC '06)*, pages 394–404, August 2006.

[12] D. R. Engler, M. F. Kaashoek, and J. W. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 251–266, Copper Mountain Resort, Colorado, December 1995.

[13] R. M. English and A. A. Stepanov. Loge: A Self-Organizing Disk Controller. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '92)*, pages 237–252, San Francisco, California, January 1992.

[14] E. Gal and S. Toledo. Algorithms and Data Structures for Flash Memories. *ACM Computing Surveys*, 37:138–163, June 2005.

[15] 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 *Proceedings of MICRO-42*, New York, New York, December 2009.

[16] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: a Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings. In *Proceedings of the 43th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*, pages 229–240, Washington, DC, March 2009.

[17] Intel Corporation. Intel X25-M Mainstream SATA Solid-State Drives. `ftp://download.intel.com/design/flash/NAND/mainstream/mainstream-sata-s%sd-datasheet.pdf`.

[18] D. Jung, Y.-H. Chae, H. Jo, J.-S. Kim, and J. Lee. A Group-based Wear-Leveling Algorithm for Large-Capacity Flash Memory Storage Systems. In *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems (CASES '07)*, October 2007.

[19] J.-U. Kang, H. Jo, J.-S. 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 (EMSOFT '08)*, Seoul, Korea, August 2006.

[20] A. Kawaguchi, S. Nishioka, and H. Motoda. A Flash-Memory Based File System. In *Proceedings of the USENIX 1995 Winter Technical Conference*, New Orleans, Louisiana, January 1995.

[21] S. Lee, D. Shin, Y.-J. Kim, and J. Kim. LAST: Locality-Aware Sector Translation for NAND Flash Memory-Based Storage Systems. *In Proceedings of the International Workshop on Storage and I/O Virtualization, Performance, Energy, Evaluation and Dependability (SPEED2008)*, February 2008.

[22] 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. *IEEE Transactions on Embedded Computing Systems*, 6, 2007.

[23] P. Macko, M. Seltzer, and K. A. Smith. Tracking Back References in a Write-Anywhere File System. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST '10)*, San Jose, California, February 2010.

[24] 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 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 238–251, Saint-Malo, France, October 1997.

[25] A. One. YAFFS: Yet Another Flash File System, 2002. `http://www.yaffs.net/`.

[26] 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 the Management of Data (SIGMOD '88), pages 109–116, Chicago, Illinois, June 1988.

[27] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[28] D. Spinellis. Another Level of Indirection. In A. Oram and G. Wilson, editors, *Beautiful Code: Leading Programmers Explain How They Think*, chapter 17, pages 279–291. O'Reilly and Associates, 2007.

[29] Sun Microsystems. Solaris Internals: FileBench. `http://www.solarisinternals.com/wiki/index.php/FileBench`.

[30] R. Wang, T. E. Anderson, and D. A. Patterson. Virtual Log-Based File Systems for a Programmable Disk. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, New Orleans, Louisiana, February 1999.

[31] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[32] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.

[33] D. Woodhouse. JFFS2: The Journalling Flash File System, Version 2, 2001. `http://sources.redhat.com/jffs2/jffs2`.

The Bleak Future of NAND Flash Memory

Laura M. Grupp[*], *John D. Davis*[†], *Steven Swanson*[*]
[*]*Department of Computer Science and Engineering, University of California, San Diego*
[†]*Microsoft Research, Mountain View*

Abstract

In recent years, flash-based SSDs have grown enormously both in capacity and popularity. In high-performance enterprise storage applications, accelerating adoption of SSDs is predicated on the ability of manufacturers to deliver performance that far exceeds disks while closing the gap in cost per gigabyte. However, while flash density continues to improve, other metrics such as a reliability, endurance, and performance are all declining. As a result, building larger-capacity flash-based SSDs that are reliable enough to be useful in enterprise settings and high-performance enough to justify their cost will become challenging.

In this work, we present our empirical data collected from 45 flash chips from 6 manufacturers and examine the performance trends for these raw flash devices as flash scales down in feature size. We use this analysis to predict the performance and cost characteristics of future SSDs. We show that future gains in density will come at significant drops in performance and reliability. As a result, SSD manufacturers and users will face a tough choice in trading off between cost, performance, capacity and reliability.

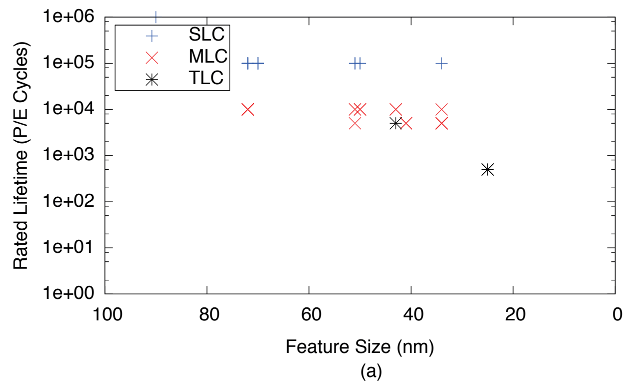
1 Introduction

Flash-based Solid State Drives (SSDs) have enabled a revolution in mobile computing and are making deep inroads into data centers and high-performance computing. SSDs offer substantial performance improvements relative to disk, but cost is limiting adoption in cost-sensitive applications and reliability is limiting adoption in higher-end machines. The hope of SSD manufactures is that improvements in flash density through silicon feature size scaling (shrinking the size of a transistor) and storing more bits per storage cell will drive down costs and increase their adoption. Unfortunately, trends in flash technology suggest that this is unlikely.

While flash density in terms of bits/mm^2 and feature size scaling continues to increase rapidly, all other figures of merit for flash – performance, program/erase endurance, energy efficiency, and data retention time – decline steeply as density rises. For example, our data show each additional bit per cell increases write latency by $4\times$ and reduces program/erase lifetime by $10\times$ to $20\times$ (as shown in Figure 1), while providing decreasing returns in density ($2\times$, $1.5\times$, and $1.3\times$ between 1-,2-,3- and 4-bit cells, respectively). As a result, we are reaching the limit of what current flash management techniques can deliver in terms of usable capacity – we may be able to build more spacious SSDs, but they may be too slow and unreliable to be competitive against disks of similar cost in enterprise applications.

This paper uses empirical data from 45 flash chips manufactured by six different companies to identify trends in flash technology scaling. We then use those trends to make projections about the performance and cost of future SSDs. We construct an idealized SSD model that makes optimistic assumptions about the efficiency of the flash translation layer (FTL) and shows that as flash continues to scale, it will be extremely difficult to design SSDs that reduce cost per bit without becoming either too slow or too unreliable (or both) as to be unusable in enterprise settings. We conclude that the cost per bit for enterprise-class SSDs targeting general-purpose applications will stagnate.

The rest of this paper is organized as follows. Section 2 outlines the current state of flash technology. Section 3 describes the architecture of our idealized SSD design, and how we combine it with our measurements to project the behavior of future SSDs. Section 4 presents the results of this idealized model, and Section 5 concludes.



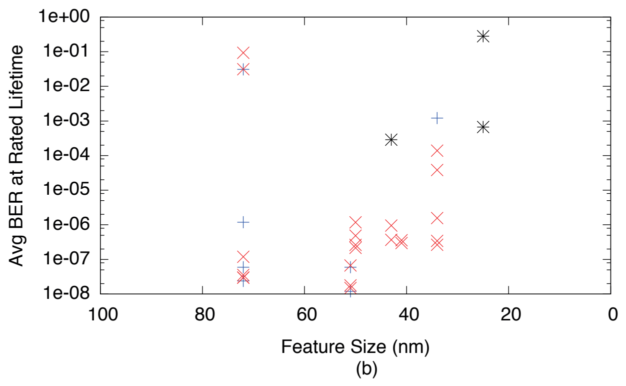


Figure 1: **Trends in Flash's Reliability** Increasing flash's density by adding bits to a cell or by decreasing feature size reduces both (a) lifetime and (b) reliability.

2 The State of NAND Flash Memory

Flash-based SSDs are evolving rapidly and in complex ways – while manufacturers drive toward higher densities to compete with HDDs, increasing density by using newer, cutting edge flash chips can adversely affect performance, energy efficiency and reliability.

To enable higher densities, manufacturers scale down the manufacturing feature size of these chips while also leveraging the technology's ability to store multiple bits in each cell. Most recently on the market are 25 nm cells which can store three bits each (called Triple Level Cells, or *TLC*). Before TLC came 2-bit, multi-level cells (MLC) and 1-bit single-level cells (SLC). Techniques that enable four or more bits per cell are on the horizon [12].

Figure 2, collects the trend in price of raw flash memory from a variety of industrial sources, and shows the drop in price per bit for the higher density chips. Historically, flash cost per bit has dropped by between 40 and 50% per year [3]. However, over the course of 2011, the price of flash flattened out. If flash has trouble scaling beyond 12nm (as some predict), the prospects for further cost reductions are uncertain.

The limitations of MLC and TLC's reliability and performance arise from their underlying structures. Each flash cell comprises a single transistor with an added layer of metal between the gate and the channel, called the floating gate. To change the value stored in the cell, the program operation applies very high voltages to its terminals which cause electrons to tunnel through the gate oxide to reach the floating gate. To erase a cell, the voltages are reversed, pulling the electrons off the floating gate. Each of these operations strains the gate oxide, until eventually it no longer isolates the floating gate, making it impossible to store charge.

The charge on the floating gate modifies the threhold voltage, V_{TH} of the transistor (i.e. the voltage at which the transistor turns on and off). In a programmed SLC cell, V_{TH} will be in one of two ranges (since program-

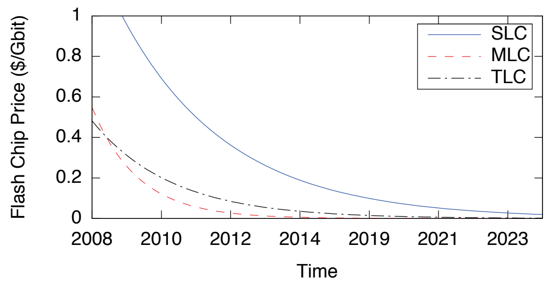


Figure 2: **Trends in Flash Prices** Flash prices reflect the target markets. Low density, SLC, parts target higher-priced markets which require more reliability while high density MLC and TLC are racing to compete with low-cost HDDs. Cameras, iPods and other mobile devices drive the low end.

ming is not perfectly precise), depending on the value the cell stores. The two ranges have a "guard band" between them. Because the SLC cell only needs two ranges and a single guard band, both ranges and the guard band can be relatively wide. Increasing the number of bits stored from one (SLC) to two (MLC) increases the number of distributions from two to four, and requires two additional guard bands. As a result, the distributions must be tighter and narrower. The necessity of narrow V_{TH} distributions increases programming time, since the chip must make more, finer adjustments to V_{TH} to program the cell correctly (as described below). At the same time, the narrow guard band reduces reliability. TLC cells make this problem even worse: They must accomodate eight V_{TH} levels and seven guard bands.

We present empirical evidence of worsening lifetime and reliability of flash as it reaches higher densities. We collected this data from 45 flash chips made by six manufacturers spanning feature sizes from 72 nm to 25 nm. Our flash characterization system (described in [4]) allows us to issue requests to a raw flash chip without FTL interference and measure the latency of each of these operations with 10 ns resolution. We repeat this program-erase cycle (P/E cycle) until each measured block reaches the rated lifetime of its chip.

Figure 1 shows the chips' rated lifetime as well as the bit error rate (BER) measured at that lifetime. The chips' lifetimes decrease slowly with feature size, but fall precipitously across SLC, MLC and TLC devices. While the error rates span a broad range, there is a clear upward trend as feature size shrinks and densities increase. Applications that require more reliable or longer-term storage prefer SLC chips and those at larger feature sizes because they experience far fewer errors for many more cycles than denser technology.

Theory and empirical evidence also indicate lower performance for denser chips, primarily for the program or write operation. Very early flash memory would apply a steady, high voltage to any cell being programed for a fixed amount of time. However, Suh et al. [10] quickly determined that the Incremental Step Pulse Programming (ISPP) would be far more effective in tolerating variation between cells and in environmental conditions. ISPP performs a series of program pulses each followed by a read-verify step. Once the cell is programmed correctly, programming for that cell stops. This algorithm is necessary because programming is a one-way operation: There is no way to "unprogram" a cell short of erasing the entire block, and overshooting the correct voltage results in storing the wrong value. ISPP remains a key algorithm in modern chips and is instrumental in improving the performance and reliability of higher-density cells.

Not long after Samsung proposed MLC for NAND flash [5, 6], Toshiba split the two bits to separate pages so that the chip could program each page more quickly by moving the cell only halfway through the voltage range with each operation [11]. Much later, Samsung provided further performance improvements to pages stored in the least significant bit of each cell [8]. By applying fast, imprecise pulses to program the fast pages, and using fine-grain, precise pulses to program the slow pages. These latter pulses generate the tight V_{TH} distributions that MLC devices require, but they make programming much slower. All the MLC and TLC devices we tested split and program the bits in a cell this way.

For SSD designers, this performance variability between pages leads to an opportunity to easily trade off capacity and performance [4, 9]. The SSD can, for example use only the fast pages in MLC parts, sacrificing half their capacity but making latency comparable to SLC. In this work, we label such a configuration "MLC-1" – an MLC device using just one bit per cell. Samsung and Micron have formalized this trade-off in multi-level flash by providing single and multi-level cell modes [7] in the same chip and we believe FusionIO uses the property in the controller of their SMLC-based drives [9].

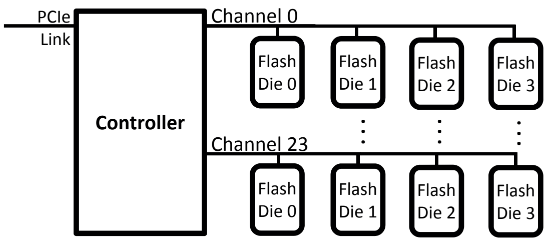


Figure 3: **Architecture of SSD-CDC** The architecture of our baseline SSD. This structure remains constant while we scale the technology used for each flash die.

Architecture Parameter	Value
Example Interface	PCIe 1.1x4
FTL Overhead Latency	30 μs
Channels	24
Channel Speed	400 MB/s [1]
Dies per Channel (DPC)	4
Baseline Parameter	**Value**
SSD Price	$7,800
Capacity	320 GB
Feature Size	34 nm
Cell Type	MLC

Table 1: **Architecture and Baseline Configuration of SSD-CDC** These parameters define the Enterprise-class, Constant Die Count SSD (SSD-CDC) architecture and starting values for the flash technology it contains.

3 A Prototypical SSD

To model the effect of evolving flash characteristics on complete SSDs we combine empirical measurement of flash chips in an SSD architecture with a constant die count called *SSD-CDC*. SSD-CDC's architecture is representative of high-end SSDs from companies such as FusionIO, OCZ and Virident. We model the complexities of FTL design by assuming optimistic constants and overheads that provide upper bounds on the performance characteristics of SSDs built with future generation flash technology.

Section 3.1 describes the architecture of SSD-CDC, while Section 3.2 describes how we combine this model with our empirical data to estimate the performance of an SSD with fixed die area.

3.1 SSD-CDC

Table 1 describes the parameters of SSD-CDC's architecture and Figure 3 shows a block representation of its architecture. SSD-CDC manages an array of flash chips and presents a block-based interface. Given current trends in PCIe interface performance, we assume that the PCIe link is not a bottleneck for our design.

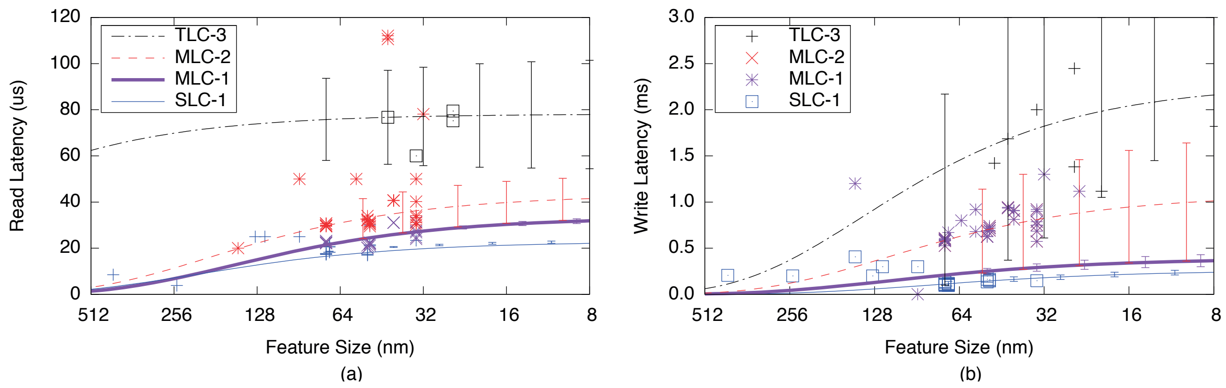


Figure 4: **Flash Chip Latency Trends** Fitting an exponential to the collection of data for each cell technology, SLC-1, MLC-1, MLC-2 and TLC-3, allows us to project the behavior of future feature sizes for (a) read latency and (b) write latency. Doing the same with one standard deviation above and below the average for each chip yields a range of probable behavior, as shown by the error bars.

Configuration	**Read Latency (μs)**		**Write Latency (μs)**	
	Equation	**-1nm**	**Equation**	**-1nm**
SLC-1	$\max = 24.0e^{-3.5e-3f}$	0.36%	$\max = 287.0e^{-1.1e-2f}$	1.07%
	$\text{avg} = 23.4e^{-3.2e-3f}$	0.32%	$\text{avg} = 262.6e^{-1.2e-2f}$	1.19%
	$\min = 22.8e^{-2.9e-3f}$	0.29%	$\min = 239.3e^{-1.3e-2f}$	1.34%
MLC-1	$\max = 34.8e^{-6.9e-3f}$	0.69%	$\max = 467.3e^{-1.0e-2f}$	1.01%
	$\text{avg} = 33.5e^{-6.3e-3f}$	0.63%	$\text{avg} = 390.0e^{-8.7e-3f}$	0.87%
	$\min = 32.2e^{-5.6e-3f}$	0.57%	$\min = 316.5e^{-7.0e-3f}$	0.70%
MLC-2	$\max = 52.5e^{-4.5e-3f}$	0.45%	$\max = 1778.2e^{-8.3e-3f}$	0.84%
	$\text{avg} = 43.3e^{-5.2e-3f}$	0.52%	$\text{avg} = 1084.4e^{-8.6e-3f}$	0.86%
	$\min = 34.2e^{-6.6e-3f}$	0.66%	$\min = 393.7e^{-9.9e-3f}$	1.00%
†**TLC-3**	$\max = 102.5e^{-1.3e-3f}$	0.13%	$\max = 4844.8e^{-1.1e-2f}$	1.12%
	$\text{avg} = 78.2e^{-4.4e-4f}$	0.04%	$\text{avg} = 2286.2e^{-7.1e-3f}$	0.71%
	$\min = 54.0e^{9.9e-4f}$	-0.10%	$\min = 2620.8e^{-4.6e-2f}$	4.67%

Table 2: **Latency Projections** We generated these equations by fitting an exponential ($y = Ae^{bf}$) to our empirical data, and they allow us to project the latency of flash as a function of feature size (f) in nm. The percentages represent the increase in latency with 1nm shrinkage. †The trends for TLC are less certain than for SLC or MLC, because our data for TLC devices is more limited.

Number	**Metric**	**Value**
1	$Capacity_{proj}$	$= Capacity_{base} \times \left(\frac{BitsPerCell_{proj}}{BitsPerCell_{base}}\right) \times \left(\frac{FeatureSize_{base}}{FeatureSize_{proj}}\right)^2$
2	SSD_BW_{proj}	$= ChannelCount \times ChannelBW_{proj}$
3	$ChannelBW_{proj}$	$= \frac{(DiesPerChannel-1)*PageSize}{DieLatency_{proj}}$, when $DieLatency_{proj} \leq BWThreshold$
4	$ChannelBW_{proj}$	$= ChannelSpeed$, when $DieLatency_{proj} > BWThreshold$
5	$TransferTime$	$= \frac{PageSize}{ChannelSpeed}$
6	$BWThreshold$	$= (DiesPerChannel - 1) \times TransferTime$
7	SSD_IOPs_{proj}	$= ChannelCount \times ChannelIOPs_{proj}$
9	$ChannelIOPs_{proj}$	$= \frac{1}{TransferTime}$, when $DieLatency_{proj} \leq IOPsThreshold$
8	$ChannelIOPs_{proj}$	$= \frac{(DiesPerChannel-1)}{DieLatency_{proj}}$, when $DieLatency_{proj} > IOPsThreshold$
10	$TransferTime$	$= \frac{AccessSize}{ChannelSpeed}$
11	$IOPsThreshold$	$= (DiesPerChannel - 1) \times TransferTime$

Table 3: **Model's Equations** These equations allow us to scale the metrics of our baseline SSD to future process technologies and other cell densities.

The SSD's controller implements the FTL. We estimate that this management layer incurs an overhead of 30 μs for ECC and additional FTL operations. The controller coordinates 24 channels, each of which connects four dies to the controller via a 400 MB/s bus. To fix the cost of SSD-CDC, we assume a constant die count equal to 96 dies.

3.2 Projections

We now describe our future projections for seven metrics of SSD-CDC: capacity, read latency, write latency, read bandwidth, write bandwidth, read IOPs and write IOPs. Table 1 provides baseline values for SSD-CDC and Table 2 summarizes the projections we make for the underlying flash technology. This section describes the formulas we use to compute each metric from the projections (summarized in Table 3). Some of the calculations involve making simplifying assumptions about SSD-CDC's behavior. In those cases, we make the assumption that maximizes the SSD's performance.

Capacity Equation 1 calculates the capacity of SSD-CDC, by scaling the capacity of the baseline by the square of the ratio of the projected feature size to the baseline feature size (34 nm). We also scale capacity depending on the number of bits per cell (BPC) the projected chip stores relative to the baseline BPC (2 – MLC). In some cases, we configure SSD-CDC to store fewer bits per cell than a projected chip allows, as in the case of MLC-1. In these cases, the projected capacity would reflect the *effective* bits per cell.

Latency To calculate the projected read and write latencies, we fit an exponential function to the empirical data for a given cell type. Figure 4 depicts both the raw latency data and the curves fitted to SLC-1, MLC-1, MLC-2 and TLC-3. To generate the data for MLC-1, which ignores the "slow" pages, we calculate the average latency for reads and writes for the "fast" pages only. Other configurations supporting reduced capacity and improved latency, such as TLC-1 and TLC-2, would use a similar method. We do not present these latter configurations, because there is very little TLC data available to create reliable predictions. Figure 4 shows each collection of data with the fitted exponentials for average, minimum and maximum, and Table 2 reports the equations for these fitted trends. We calculate the projected latency by adding the values generated by these trends to the SSD's overhead reported in Table 1.

Bandwidth To find the bandwidth of our SSD, we must first calculate each channel's bandwidth and then multiply that by the number of channels in the SSD (Equation 2). Each channel's bandwidth requires an understanding of whether channel bandwidth or per-chip

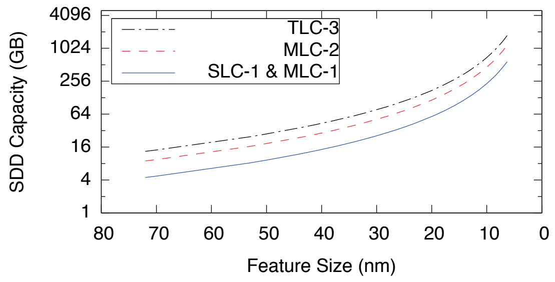


Figure 5: **Scaling of SSD Capacity** Flash manufacturers increase SSDs' capacity through both reducing feature size and storing more bits in each cell.

bandwidth is the bottleneck. Equation 6 determines the threshold between these two cases by multiplying the transfer time (see Equation 5) by one less than the number of dies on the channel. If the latency of the operation on the die is larger than this number, the die is the bottleneck and we use Equation 3. Otherwise, the channel's bandwidth is simply the speed of its bus (Equation 4).

IOPs The calculation for IOPs is very similar to bandwidth, except instead of using the flash's page size in all cases, we also account for the access size since it effects the transfer time: If the access size is smaller than one page, the system still incurs the read or write latency of one entire page access. Equations 7-11 describe the calculations.

4 Results

This section explores the performance and cost of SSD-CDC in light of the flash feature size scaling trends described above. We explore four different cell technologies (SLC-1, MLC-1, MLC-2, and TLC-3) and feature sizes scaled down from 72 nm to 6.5 nm (the smallest feature size targeted by industry consensus as published in the International Technology Roadmap for Semiconductors (ITRS) [2]), using a fixed silicon budget for flash storage.

4.1 Capacity and cost

Figure 5 shows how SSD-CDC's density will increase as the number of bits per cell rises and feature size continues to scale. Even with the optimistic goal of scaling flash cells to 6.5 nm, SSD-CDC can only achieve capacities greater than 512 GB with two or more bits per cell. TLC allows for capacities up to 1.4 TB – pushing capacity beyond this level will require more dies.

Since capacity is one of the key drivers in SSD design and because it is the only aspect of SSDs that improves consistently over time, we plot the remainder of the characteristics against SSD-CDC's capacity.

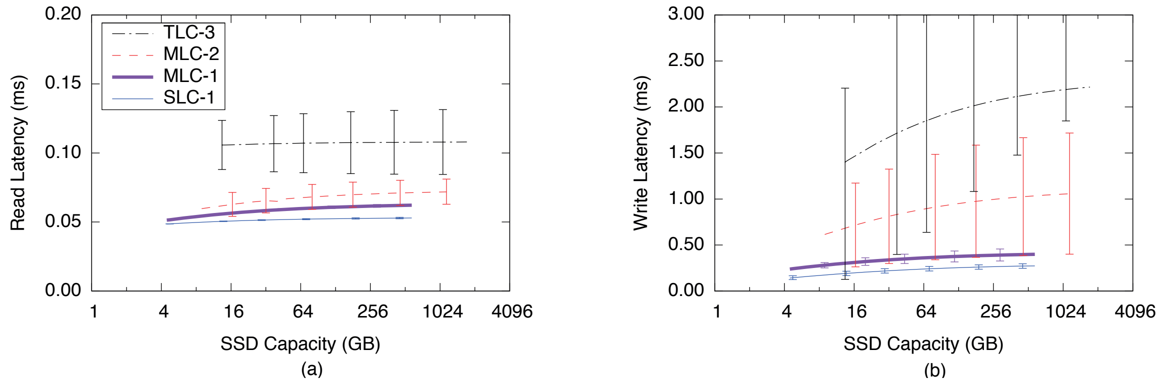


Figure 6: **SSD Latency** In order to achieve higher densities, flash manufacturers must sacrifice (a) read and (b) write latency.

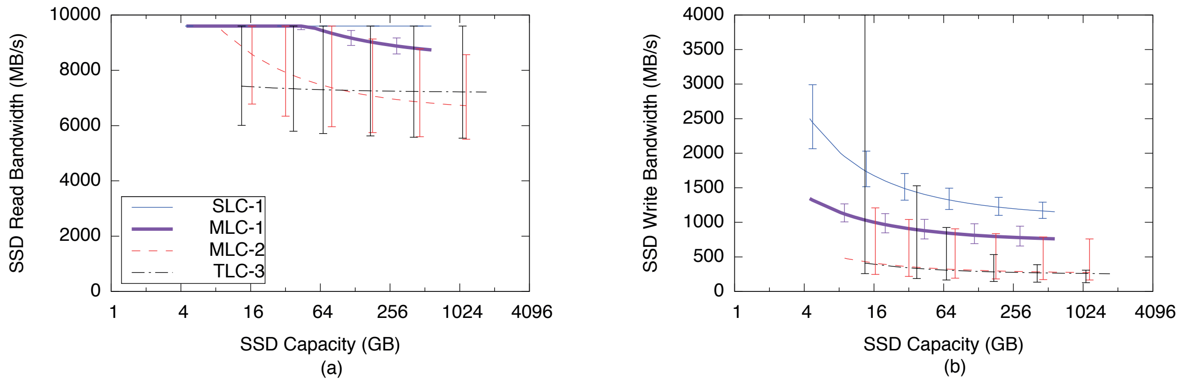


Figure 7: **SSD Bandwidth** SLC will continue to be the high performance option. To obtain higher capacities without additional dies and cost will require a significant performance hit in terms of (a) read and (b) write bandwidth moving from SLC-1 to MLC-2 or TLC-3.

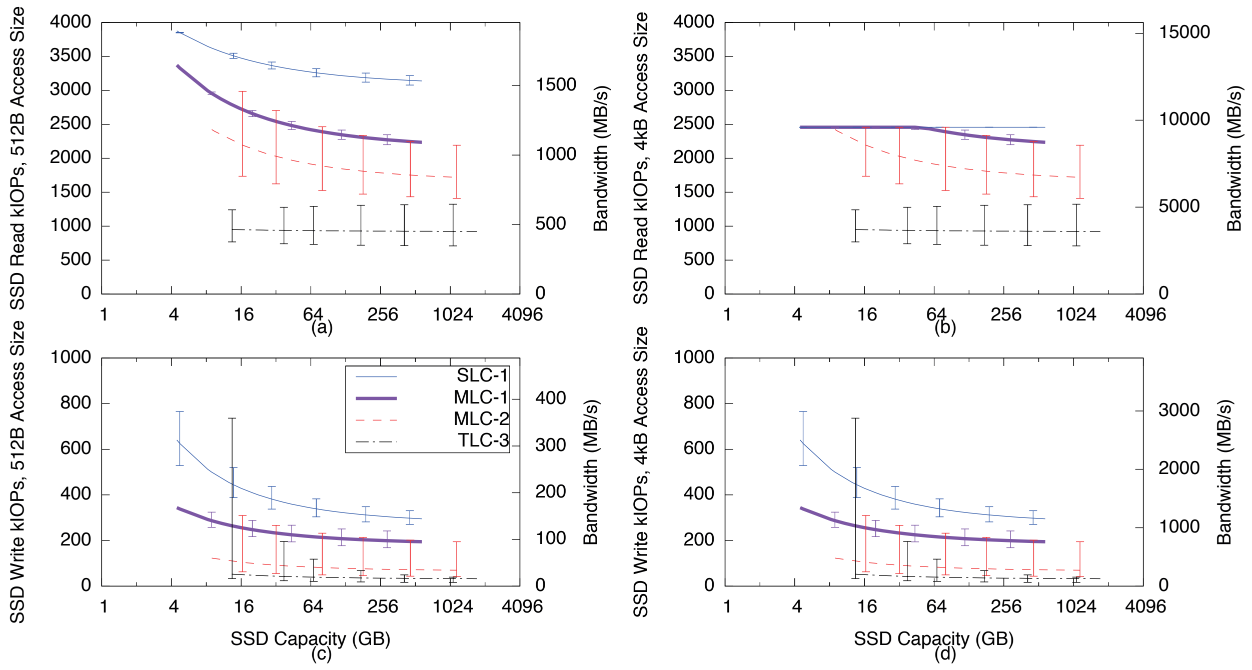


Figure 8: **SSD IOPS** With a fixed die area, higher capacities can only be achieved with low-performing MLC-2 and TLC-3 technologies, for 512B (a) reads and (c) writes and for 4kB (b) reads and (d) writes.

4.2 Latency

Reduced latency is among the frequently touted advantages of flash-based SSDs over disks, but changes in flash technology will erode the gap between disks and SSDs. Figure 6 shows how both read and write latencies increase with SSD-CDC's capacity. Reaching beyond 512 GB pushes write latency to 1 ms for MLC-2 and over 2.1 ms for TLC. Read latency, rises to least 70 μs for MLC-2 and 100 μs for TLC.

The data also makes clear the choices that SSD designers will face. Either SSD-CDC's capacity stops scaling at ~582 GB or its read and write latency increases sharply because increasing drive capacity with fixed die area would necessitate switching cell technology from SLC-1 or MLC-1 to MLC-2 or TLC-3. With current trends, our SSDs could be up to 5.5× larger, but the latency will be 2.1× worse for reads and 5.4× worse for writes. This will reduce the write latency advantage that SSDs offer relative to disk from 8.3×(vs. a 7 ms disk access) to just 3.2×. Depending on the application, this reduced improvement may not justify the higher cost of SSDs.

4.3 Bandwidth and IOPs

SSDs offer moderate gains in bandwidth relative to disks, but very large improvements in random IOP performance. However, increases in operation latency will drive down IOPs and bandwidth.

Figure 7 illustrates the effect on bandwidth. Above 128 GB or for multi-level technologies, bandwidth drops by 25% due to the latency of the program operation on the flash die.

SSDs provide the largest gains relative to disks for small, random IOPs. We present two access sizes – the historically standard disk block size of 512 B and the most common flash page size and modern disk access size of 4 kB. Figure 8 presents the performance in terms of IOPs. When using the smaller, unaligned 512B accesses, SLC and MLC chips must access 4 kB of data and the SSD must discard 88% of the accessed data. For TLC, there is even more wasted bandwidth because page size is 8 kB.

When using 4kB accesses, MLC IOPs drop as density increases, falling by 18% between the 64 and 1024 GB configurations. Despite this drop, the data suggest that SSDs will maintain an enormous (but slowly shrinking) advantage relative to disk in terms of IOPs. Even the fastest hard drives can sustain no more than 200 IOPs, and the slowest SSD configuration we consider achieves over 32,000 IOPs.

Figure 9 shows all parameters for an SSD made from MLC-2 flash normalized to SSD-CDC configured with

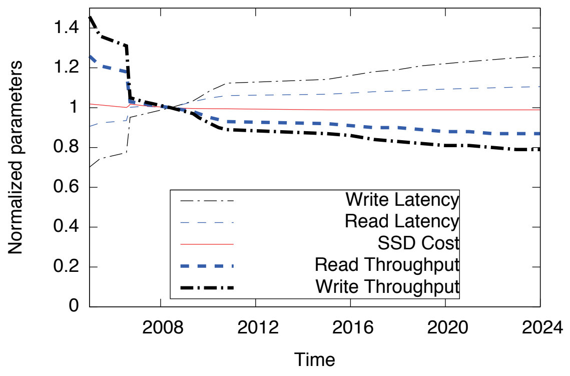


Figure 9: **Scaling of all parameters** While the cost of an MLC-based SSD remains roughly A constant, read and particularly write performance decline.

currently available flash. Our projections show that the cost of the flash in SSD-CDC will remain roughly constant and that density will continue to increase (as long as flash scaling continues as projected by the ITRS). However, they also show that access latencies will increase by 26% and that bandwidth (in both MB/s and IOPS) will drop by 21%.

5 Conclusion

The technology trends we have described put SSDs in an unusual position for a cutting-edge technology: SSDs will continue to improve by some metrics (notably density and cost per bit), but everything else about them is poised to get worse. This makes the future of SSDs cloudy: While the growing capacity of SSDs and high IOP rates will make them attractive in many applications, the reduction in performance that is necessary to increase capacity while keeping costs in check may make it difficult for SSDs to scale as a viable technology for some applications.

References

[1] Open nand flash interface specification 3.0. http://onfi.org/specifications/.

[2] International technology roadmap for semiconductors: Emerging research devices, 2010.

[3] DENALI. http://www.denali.com/wordpress/index.php/dmr/2009/07/16/nand-forward-prices-rate-of-decline-will.

[4] GRUPP, L. M., CAULFIELD, A. M., COBURN, J., SWANSON, S., YAAKOBI, E., SIEGEL, P. H., AND WOLF, J. K. Characterizing flash memory: anomalies, observations, and applications. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2009), ACM, pp. 24–33.

[5] JUNG, T.-S., CHOI, Y.-J., SUH, K.-D., SUH, B.-H., KIM, J.-K., LIM, Y.-H., KOH, Y.-N., PARK, J.-W., LEE, K.-J., PARK, J.-H., PARK, K.-T., KIM, J.-R., LEE, J.-H., AND LIM, H.-K. A 3.3 v 128 mb multi-level nand flash memory for mass storage applications. In *Solid-State Circuits Conference, 1996. Digest*

of Technical Papers. 42nd ISSCC., 1996 IEEE International (feb 1996), pp. 32 –33, 412.

[6] Jung, T.-S., Choi, Y.-J., Suh, K.-D., Suh, B.-H., Kim, J.-K., Lim, Y.-H., Koh, Y.-N., Park, J.-W., Lee, K.-J., Park, J.-H., Park, K.-T., Kim, J.-R., Yi, J.-H., and Lim, H.-K. A 117-mm2 3.3-v only 128-mb multilevel nand flash memory for mass storage applications. *Solid-State Circuits, IEEE Journal of 31*, 11 (nov 1996), 1575 –1583.

[7] Marotta, G. e. a. A 3bit/cell 32gb nand flash memory at 34nm with 6mb/s program throughput and with dynamic 2b/cell blocks configuration mode for a program throughput increase up to 13mb/s. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International* (feb. 2010), pp. 444 –445.

[8] Park, K.-T., Kang, M., Kim, D., Hwang, S.-W., Choi, B. Y., Lee, Y.-T., Kim, C., and Kim, K. A zeroing cell-to-cell interference page architecture with temporary lsb storing and parallel msb program scheme for mlc nand flash memories. *Solid-State Circuits, IEEE Journal of 43*, 4 (april 2008), 919 –928.

[9] Raffo, D. Fusionio builds ssd bridge between slc,mlc, july 2009.

[10] Suh, K.-D., Suh, B.-H., Lim, Y.-H., Kim, J.-K., Choi, Y.-J., Koh, Y.-N., Lee, S.-S., Kwon, S.-C., Choi, B.-S., Yum, J.-S., Choi, J.-H., Kim, J.-R., and Lim, H.-K. A 3.3 v 32 mb nand flash memory with incremental step pulse programming scheme. *Solid-State Circuits, IEEE Journal of 30*, 11 (nov 1995), 1149 –1156.

[11] Takeuchi, K., Tanaka, T., and Tanzawa, T. A multipage cell architecture for high-speed programming multilevel nand flash memories. *Solid-State Circuits, IEEE Journal of 33*, 8 (aug 1998), 1228 –1238.

[12] Trinh, C. e. a. A 5.6mb/s 64gb 4b/cell nand flash memory in 43nm cmos. In *Solid-State Circuits Conference - Digest of Technical Papers, 2009. ISSCC 2009. IEEE International* (feb. 2009), pp. 246 –247,247a.

When Poll is Better than Interrupt

Jisoo Yang Dave B. Minturn Frank Hady

{jisoo.yang | dave.b.minturn | frank.hady} (at) intel.com

Intel Corporation

Abstract

In a traditional block I/O path, the operating system completes virtually all I/Os asynchronously via interrupts. However, performing storage I/O with ultra-low latency devices using next-generation non-volatile memory, it can be shown that polling for the completion – hence wasting clock cycles during the I/O – delivers higher performance than traditional interrupt-driven I/O. This paper thus argues for the *synchronous completion* of block I/O first by presenting strong empirical evidence showing a stack latency advantage, second by delineating limits with the current interrupt-driven path, and third by proving that synchronous completion is indeed safe and correct. This paper further discusses challenges and opportunities introduced by synchronous I/O completion model for both operating system kernels and user applications.

1 Introduction

When an operating system kernel processes a block storage I/O request, the kernel usually submits and completes the I/O request asynchronously, releasing the CPU to perform other tasks while the hardware device completes the storage operation. In addition to the CPU cycles saved, the asynchrony provides opportunities to reorder and merge multiple I/O requests to better match the characteristics of the backing device and achieve higher performance. Indeed, this asynchronous I/O strategy has worked well for traditional rotating devices and even for NAND-based solid-state drives (SSDs).

Future SSD devices may well utilize high-performance next-generation non-volatile memory (NVM), calling for a re-examination of the traditional asynchronous completion model. The high performance of such devices both diminish the CPU cycles saved by asynchrony and reduce the I/O scheduling advantage.

This paper thus argues for the *synchronous I/O completion model* by which the kernel path handling an I/O request stays within the process context that initiated the I/O. Synchronous completion allows I/O requests to bypass the kernel's heavyweight asynchronous block I/O subsystem, reducing CPU clock cycles needed to process I/Os. However, a necessary condition is that the CPU has to spin-wait for the completion from the device, increasing the cycles used.

Using a prototype DRAM-based storage device to mimic the potential performance of a very fast next-generation SSD, we verified that the synchronous model completes an individual I/O faster and consumes less CPU clock cycles despite having to poll. The device is fast enough that the spinning time is smaller than the overhead of the asynchronous I/O completion model.

Interrupt-driven asynchronous completion introduces additional performance issues when used with very fast SSDs such as our prototype. Asynchronous completion may suffer from lower I/O rates even when scaled to many outstanding I/Os across many threads. We empirically confirmed this with Linux,* and examine the system overheads of interrupt handling, cache pollution, CPU power-state transitions associated with the asynchronous model.

We also demonstrate that the synchronous completion model is correct and simple with respect to maintaining I/O ordering when used with application interfaces such as non-blocking I/O and multithreading.

We suggest that current applications may further benefit from the synchronous model by avoiding the non-blocking storage I/O interface and by reassessing buffering strategies such as I/O prefetching. We conclude that with future SSDs built of next-generation NVM elements, introducing the synchronous completion model could reap significant performance benefits.

2 Background

The commercial success of SSDs coupled with reported advancements of NVM technology is significantly reducing the performance gap between mass-storage and memory [15]. Experimental storages device that complete an I/O within a few microseconds have been demonstrated [8]. One of the implications of this trend is that the once negligible cost of I/O stack time becomes more relevant [8,12]. Another important trend in operating with SSDs is that big, sequential, batched I/O requests need no longer be favored over small, random I/O requests [17].

In the traditional block I/O architecture, the operating system's *block I/O subsystem* performs the task of scheduling I/O requests and forwarding them to block device drivers. This subsystem processes *kernel I/O requests* specifying the starting disk sector, target memory address, and size of I/O transfer, and originating from a file system, page cache, or user application using direct I/O. The block I/O subsystem schedules kernel I/O requests by queueing them in a *kernel I/O queue* and placing the I/O-issuing thread in an I/O wait state. The queued requests are later forwarded to a low-level block device driver, which translates the requests into *device I/O commands* specific to the backing storage device.

Upon finishing an I/O command, a storage device is expected to raise a hardware interrupt to inform the device driver of the completion of a previously submitted command. The device driver's interrupt service routine then notifies the block I/O subsystem, which subsequently ends the kernel I/O request by releasing the target memory and un-blocking the thread waiting on the completion of the request. A storage device may handle multiple device commands concurrently using its own *device queue* [2,5,6], and may combine multiple completion interrupts, a technique called interrupt coalescing to reduce overhead.

As described the traditional block I/O subsystem uses asynchrony within the I/O path to save CPU cycles for other tasks while the storage device handles I/O commands. Also, using I/O schedulers, the kernel can reorder or combine multiple outstanding kernel I/O requests to better utilize the underlying storage media.

This description of the traditional block storage path captures what we will refer to as the *asynchronous I/O completion model*. In this model, the kernel submits a device I/O command in a context distinct from the context of the process that originated the I/O. The hardware interrupt generated by the device upon command completion is also handled, at first, by a separate kernel context. The original process is later awakened to resume its execution.

A block I/O subsystem typically provides a set of in-kernel interfaces for a device driver use. In Linux, a block device driver is expected to implement a 'request_fn' callback that the kernel calls while executing in an interrupt context [7,10]. Linux provides another callback point called 'make_request', which is intended to be used by pseudo block devices, such as a ramdisk. The latter callback differs from the former one in that the latter is positioned at highest point in the Linux's block I/O subsystem and called within the context of the process thread.

3 Synchronous I/O completion model

When we say a process completes an I/O synchronously, we mean the kernel's entire path handling an I/O request stays within the process context that initiated the I/O. A necessary condition for this *synchronous I/O completion* is that the CPU poll the device for completion. This polling must be realized by a spin loop, busy-waiting the CPU while waiting for the completion.

Compared to the traditional asynchronous model, synchronous completion can reduce CPU clock cycles needed for a kernel to process an I/O request. This reduction comes primarily from a shortened kernel path and from the removal of interrupt handling, but synchronous completion brings with it an extra clock cycles spent in polling. In this section, we make the case for the synchronous completion by quantifying these overheads. We then discuss problems with the asynchronous model and argue the correctness of synchronous model.

3.1 Prototype hardware and device driver

For our measurements, we used a DRAM-based prototype block storage device connected to the system with an early prototype of an NVM Express* [5] interface to serve as a model of a fast future SSD based on next-generation NVM. The device was directly attached to PCIe* Gen2 bus with eight lanes and with a device-based DMA engine handling data transfers. As described by the NVM Express specification the device communicates with the device driver via segments of main memory, through which the device receives commands and places completions. The device can instantiate multiple device queues and can be configured to generate hardware interrupts upon command completion.

I/O completion method	512B xfer	4KiB xfer
Interrupt, Gen2 bus, enters C-state	3.3 μs	4.6 μs
Interrupt, Gen2 bus	2.6 μs	4.1 μs
Polling, Gen2 bus	1.5 μs	2.9 μs
Interrupt, 8Gbps bus projection	2.0 μs	2.6 μs
Polling, 8Gbps bus projection	0.9 μs	1.5 μs

Table 1. Time to finish an I/O command, excluding software time, measured for our prototype device. The numbers measure random-read performance with device queue depth of 1.

Table 1 shows performance statistics for the prototype device. The 'C-state' refers to the latency when the CPU enters power-saving mode while the I/O is outstanding. The performance measured is limited by prototype throughput, not by anything fundamental, future SSDs may well feature higher throughputs. The improved performance projection assumes a higher throughput SSD on a saturated PCIe Gen3 bus (8Gbps).

We wrote a Linux device driver for the prototype hardware supporting both asynchronous and synchronous completion models. For the asynchronous model the driver implements Linux's 'request_fn' callback, thus taking the traditional path of using the stock kernel I/O queue. In this model, the driver uses a hardware interrupt. The driver executes within the interrupt context for both the I/O request submission and the completion. For the synchronous model, the driver implements Linux's 'make_request' callback, bypassing most of the Linux's block I/O infrastructure. In this model the driver polls for completion from device and hence executes within the context of the thread that issued the I/O.

For this study, we assume that hardware never triggers internal events that incur substantially longer latency than average. We expect that such events are rare and can be easily dealt with by having operating system fall back to traditional asynchronous model.

3.2 Experimental setup and methodology

We used 64bit Fedora* 13 running 2.6.33 kernel on an x86 dual-socket server with 12GiB of main memory. Each processor socket was populated with quad-core 2.93GHz Intel® Xeon® with 8MiB of shared L3 cache and 256KiB of per-core L2 cache. Intel® Hyper-Threading Technology was enabled totaling 16 architectural CPUs available to software. CPU frequency-scaling was disabled.

For measurements we used a combination of the CPU timestamp counter and reports from user-level programs. Upon events of interest in kernel, the device driver executed the 'rdtsc' instruction to read the CPU timestamp counter, whose values were later processed offline to produce kernel path latencies. For application IOPS (I/O Operations Per Second) and I/O system call completion latency, we used the numbers reported by 'fio' [1] I/O micro-benchmark running in user mode.

We bypassed the file system and the buffer cache to isolate the cost of the block I/O subsystem. Note that our objective is to measure the difference between the two completion models when exercising the back-end block I/O subsystem whose performance is not changed by the use of the file system or the buffer cache and would thus be additive to either completion model. The kernel was compiled with -O3 optimization and kernel preemption was enabled. The I/O scheduler was disabled for the asynchronous path by selecting 'noop' scheduler in order to make the asynchronous path as fast as possible.

3.3 Storage stack latency comparison

Our measurement answers following questions:

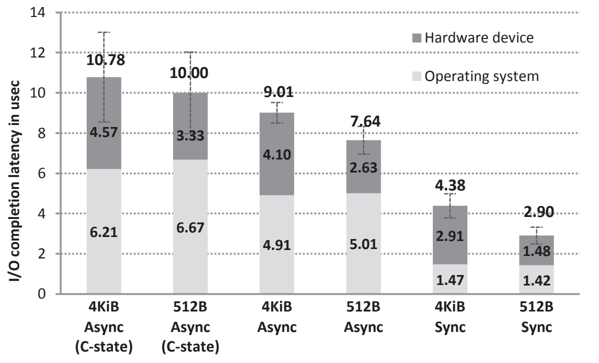


Figure 1. Storage stack block I/O subsystem cost comparison. Each bar measures application-observed I/O completion latency, which is broken into device hardware latency and non-overlapping operating system latency. Error bars represent +/- one standard deviation.

- How fast does each completion path complete application I/O requests?
- How much CPU time is spent by the kernel in each completion model?
- How much CPU time is available to another user process scheduled in during an asynchronous I/O?

Figure 1 shows that the synchronous model completes an I/O faster than asynchronous path in terms of absolute latency. The figure shows actual measured latency for the user application performing 4KiB and 512B random reads. For our fast prototype storage device the CPU spin-wait cost in the synchronous path is lower than the code-path reduction achieved by the synchronous path, completing a 4KiB I/O synchronously in 4.4μs versus 7.6μs for the asynchronous case. The figure breaks the latency into hardware time and non-hardware overlapping kernel time. The hardware time for the asynchronous path is slightly greater than that of the synchronous path due to interrupt delivery latency.

Figure 2 details the latency component breakdown of the asynchronous kernel path. In the figure, *Tu* indicates the CPU time actually available to another user process during the time slot vacated during asynchronous path I/O completion. To measure this time as accurately as possible, we implemented a separate user-level program scheduled to run on the same CPU as the I/O benchmark. This program continuously checked CPU timestamps to detect its scheduled period at a sub-microsecond granularity. Using this program, we measured *Tu* to be 2.7μs with 4KiB transfer that the device takes 4.1μs to finish.

The conclusion of the stack latency measurements is a strong one: the synchronous path completes I/Os faster and more efficiently uses the CPU. This is true despite spin-waiting for the duration of the I/O because the work the CPU performs in asynchronous path (i.e., *Ta* + *Tb* =

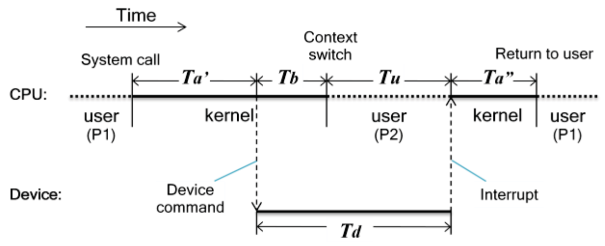


Figure 2. Latency component breakdown of asynchronous kernel path. *Ta* (= *Ta'* + *Ta"*) indicates the cost of kernel path that does not overlap with *Td*, which is the interval during which the device is active. Scheduling a user process P2 during the I/O interval incurs kernel scheduling cost, which is *Tb*. The CPU time available for P2 to make progress is *Tu*. For a 4KiB transfer, *Ta*, *Td*, *Tb*, and *Tu* measure 4.9, 4.1, 1.4 and 2.7μs, respectively.

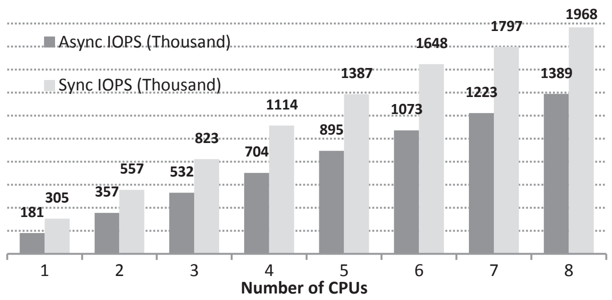


Figure 3. Scaling of storage I/Os per second (IOPS) with increased number of CPUs. For asynchronous IOPS, I/O threads are added until the utilization of each CPU reaches 100%.

6.3μs) is greater than the spin-waiting time of the synchronous path (4.38μs) with this fast prototype SSD. For smaller-sized transfers, synchronous completion by polling wins over asynchronous completion by an even greater margin.

With the synchronous completion model, improvement in hardware latency directly translates to improvement in software stack overhead. However, the same does not hold for the asynchronous model. For instance, using projected PCIe Gen3 bus performance, the spin-wait time is expected to be reduced from current 2.9μs to 1.5μs, making the synchronous path time be 3.0μs, while the asynchronous path overhead remains the same at 6.3μs. Of course the converse is also true, slow SSDs will be felt by the synchronous model, but not by the asynchronous model – clearly these results are most relevant for very low latency NVM.

This measurement study also sets a lower bound on the SSD latency for which the asynchronous completion model recovers absolutely no useful time for other processes: 1.4μs (*Tb* in Figure 2).

3.4 Further issues with interrupt-driven I/O

The increased stack efficiency gained with the synchronous model for low latency storage devices does not just result in lower latency, but also in higher IOPS. Figure 3 shows the IOPS scaling for increasing number of CPUs performing 512B randomly addressed reads. For this test, both the synchronous and asynchronous models use 100% of each included CPU. The synchronous model does so with just a single thread per CPU, while the asynchronous model required up to 8 threads per CPU to achieve maximum IOPS. In the asynchronous model, the total number of threads needed increases with number of processors to compensate for the larger per-I/O latency.

The synchronous model shows the best per-CPU I/O performance, scaling linearly with the increased number of CPUs up to 2 million IOPS – the hardware limitation of our prototype device. Even with its larger number of threads per CPU, the asynchronous model displays a significantly lower I/O rate, achieving only 60-70% of the synchronous model. This lower I/O rate is a result of inefficiencies inherent in the use of the asynchronous model when accessing such a low latency storage device. We discuss these inefficiencies in the following sections. It should be noted that this discussion is correct only for a very low latency storage device, like the one used here: traditional higher latency storage devices gain compelling efficiencies from the use the asynchronous model.

Interrupt overhead

The asynchronous model necessarily includes generation and service of an interrupt. This interrupt brings with it extra, otherwise unnecessary work increasing CPU utilization and therefore decreasing I/O rate on a fully loaded system. Another problem is that the kernel processes hardware interrupts at high priority. Our prototype device can deliver hundreds of thousands interrupts per second. Even if the asynchronous model driver completes multiple outstanding I/Os during a single hardware interrupt invocation, the device generates interrupts fast enough to saturate the system and cause user noticeable delays. Further while coalescing interrupts reduces CPU utilization overhead, it also increases completion latencies for individual I/Os.

Cache and TLB pollution

The short I/O-wait period in asynchronous model can cause a degenerative task schedule, polluting hardware cache and TLBs. This is because the default task scheduler eagerly finds any runnable thread to fill in the slot vacated by an I/O. With our prototype, the available time for a schedule in thread is only 2.7μs, which equals 8000 CPU clock cycles. If the thread scheduled is lower priority than the original thread, the original thread will likely be re-scheduled upon the completion of the I/O – lots of state swapping for little work done. Worse, thread data held in hardware resources such as memory cache and TLBs are replaced, only to be re-populated again when the original thread is scheduled back.

CPU power-state complications

Power management used in conjunction with the asynchronous model for the short I/O-wait of our device may not only reduce the power saving, but also increase I/O completion latency. A modern processor may enter a power-saving ‘C-state’ when not loaded or lightly loaded. Transition among C-states incurs latency. For the asynchronous model, the CPU enters into a power saving C-state when the scheduler fails to find a thread to run after sending an I/O command. The synchronous model does not automatically allow this transition to a lower C-state since the processor is busy.

We have measured a latency impact from C-state transition. When the processor enters into a C-state, the asynchronous path takes an additional 2μs in observed hardware latency with higher variability (Figure 1, labeled ‘async C-state’). This additional latency is incurred only when the system has no other thread to schedule on the CPU. The end result is that a thread performing I/Os runs *slower* when it is the only thread active on the CPU – we confirmed this empirically.

It is hard for an asynchronous model driver to fine-tune C-state transitions. In asynchronous path, the C-state transition decision is primarily made by operating system’s CPU scheduler or by the processor hardware itself. On the other hand, a device driver using synchronous completion can directly construct its spin-wait loop using instructions with power-state hints, such as mwait [3], better controlling C-state transitions.

3.5 Correctness of synchronous model

A block I/O subsystem is deemed correct when it preserves ordering requirements for I/O requests made by its frontend clients. Ultimately, we want to address the following problem:

> A client performs I/O calls ‘*A*’ and ‘*B*’ in order, and its ordering requirement is that *B* should get to the device after *A*. Does synchronous model respect this requirement?

For brevity, we assume that the client to be a user application using Linux I/O system calls. We also assume a file system and the page cache are bypassed. In fact, file system and page cache themselves can be considered as frontend clients using the block I/O subsystem.

We start with two assumptions:

A1. Application uses blocking I/O system calls.

A2. Application is single threaded.

Let us consider a single thread is submitting *A* and *B* in order. The operating system may preempt and schedule the thread on a different CPU, but it does not affect the ordering of I/O requests since there is only a single thread of execution. Therefore, it is guaranteed that *B* reaches to the device after *A*.

Let us relax A1. The application order requires the thread to submit *A* before *B* using non-blocking interface or AIO [4]. With the synchronous model, this means that the device has already completed the I/O for *A* at the moment that the application makes another non-blocking system calls for *B*. Therefore, the synchronous model guarantees that *B* reaches to the device after *A* with non-blocking I/O interface.

Relaxing A2, let us imagine two threads T1 and T2, each performing *A* and *B* respectively. In order to respect the application’s ordering requirement, T2 must synchronize with T1 to avoid a race in such a way that T2 must wait for T1 before submitting *B*. The end result is that the kernel always sees *B* after kernel safely completes previously submitted *A*. Therefore, the synchronous model guarantees the ordering with multi-threaded applications.

The above exercise shows that an I/O barrier is unnecessary in the synchronous model to guarantee I/O ordering. This contrasts with asynchronous model where a program has to rely on an I/O barrier when it needs to force ordering. Hence, synchronous model has a potential to further simplify storage I/O routines with respect to guaranteeing data durability and consistency.

Our synchronous device driver written for Linux has been tested with multi-threaded applications using non-blocking system calls. For instance, the driver has withstood many hours of TPC-C* benchmark run. The driver has also been heavily utilized as a system swap space. We believe that the synchronous completion model is correct and fully compatible with existing applications.

4 Discussion

The asynchronous model may work better in processing I/O requests with large transfer sizes or handling hardware stalls that cause long latencies. Hence, a favorable solution would be a synchronous and asynchronous hybrid, where there are two kernel paths for a block device: the synchronous path is the fast path for small transfers and often used, whereas the asynchronous path is the slow fallback path for large transfers or hardware stalls.

We believe that existing applications have primarily assumed the asynchronous completion model and traditional slow storage devices. Although the synchronous completion model requires little change to existing software to run correctly, some changes to the operating system and to applications will allow for faster, more efficient system operation when storage is used synchronously. We did not attempt to re-write applications, but do suggest possible software changes.

Perhaps the most significant improvement that could be achieved for I/O intensive applications is to avoid using the non-blocking user I/O interface such as AIO calls when addressing a storage device synchronously. In this case, using the non-blocking interface adds overhead and complexity to the application without benefit because operating system already completes the I/O upon the return from a non-blocking I/O submission call. Although applications that use the non-blocking interface are functionally safe and correct with synchronous completion, the use of non-blocking interface negates the latency and scalability gains achievable in kernel with the synchronous completion model.

When the backing storage device is fast enough to complete an I/O synchronously, applications that have traditionally self-managed I/O buffers must reevaluate their buffering strategy. We observe that many I/O intensive applications existing today, such as databases, the operating system's page cache, and disk-swap algorithms, employ elaborate I/O buffering and prefetching schemes. Such custom I/O schemes may add overhead with little value for the synchronous completion model. Although our work in the synchronous model greatly simplifies I/O processing overhead in the kernel, application complexity may still become a bottleneck. For instance, I/O prefetching becomes far less effective and could even hurt performance. We have found the performance of page cache and disk-swapper to increase when we disabled page cache read-ahead and swap-in clustering.

Informing applications of the presence of synchronous completions is therefore necessary. For example, an ioctl() extension to query underlying completion model should help applications decide the best I/O strategy. Operating system processor usage statistics must account separately for the time spent at the driver's spin-wait loop. Currently there is no accepted method of accounting for this 'spinning I/O wait' cycles. In our prototype implementation, the time spent in the polling loop is simply accounted towards system time. This may mislead people to believe no I/O has been performed or to suspect kernel inefficiency due to increased system time.

5 Related work

Following the success of NAND-based storage, research interest has surged on the next-generation non-volatile memory (NVM) elements [11,14,16,19]. Although base materials differ, these memory elements commonly promise faster and simpler media access than NAND.

Because of the DRAM-like random accessibility of many next-generation NVM technologies, there is abundant research in storage-class memories (SCM), where NVM is directly exposed as a physical address space. For instance, file systems have been proposed on SCM-based architectures [9,21]. In contrast, we approach next-generation NVM in a more evolutionary way, preserving the current hardware and software storage interface, in keeping with the huge body of existing applications.

Moneta [8] is a recent effort to evaluate the design and impact of next-generation NVM-based SSDs. Moneta hardware is akin to our prototype device in spirit because it is a block device connected via PCIe bus. But implementation differences enabled our hardware to perform faster than Moneta. Moneta also examined spinning to cut the kernel cost, but its description is limited to latency aspect. In contrast, this paper studied issues relevant to the viability of synchronous completion, such as IOPS scalability, interrupt thrashing, power state, etc.

Interrupt-driven asynchronous completion has long been the only I/O model used by kernel to perform real storage I/Os. Storage interface standards have thus embraced hardware queueing techniques that further improve performance of asynchronous I/O operations [2,5,6]. However, these are mostly effective for the devices with slower storage medium such as hard disk or NAND flash.

It is a well-known strategy to choose a poll-based waiting primitive over an event-based one when the waiting time is short. A spinlock, for example, is preferred to a system mutex lock if the duration of the lock is held is short. Another example is the optional use of polling [18,20] for network message passing among nodes when implementing the MPI* library [13] used in high-performance computing clusters. In such systems communication latencies among nodes are just several microseconds due to the use of low-latency, high-bandwidth communication fabric along with a highly optimized network stack such as Remote Direct Memory Access (RDMA*).

6 Conclusion

This paper makes the case for the synchronous completion of storage I/Os. When performing storage I/O with ultra-low latency devices employing next-generation non-volatile memories, polling for completion performs better than the traditional interrupt-driven asynchronous I/O path. Our conclusion has a practical importance, pointing to the need for kernel researchers to consider optimizations to the traditional kernel block storage interface with next-generation SSDs, built of next-generation NVM elements in mind. It is our belief that non-dramatic changes can reap significant benefit.

Acknowledgements

We thank members of Storage Technology Group in Intel Corporation for supporting this work. We also thank our shepherd David Patterson and the anonymous reviewers for their detailed feedback and guidance. The views and conclusions in this paper are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Intel Corporation.

References

[1] Jen Axboe. Flexible I/O tester (fio). http://git.kernel.dk/?p=fio.git;a=summary. 2010.

[2] Amber Huffman and Joni Clark. Serial ATA native command queueing. Technical white paper, http://www.seagate.com/content/pdf/whitepaper/D2c_tech_paper_intc-stx_sata_ncq.pdf, July 2003.

[3] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1-3*. Intel, 2008.

[4] M. Tim Jones. Boost application performance using asynchronous I/O. http://www.ibm.com/developerworks/linux/library/l-async/, 2006.

[5] NVMHCI Work Group. NVM Express. http://www.nvmexpress.org/, 2011.

[6] SCSI Tagged Command Queueing, SCSI Architecture Model – 3, 2007.

[7] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*, 3rd Ed., O'Reilly, 2005.

[8] Adrian M. Caufield, Arup De, Joel Coburn, Todor I. Mollov, Rajesh K. Gupta, and Steven Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories, In *Proceedings of the 43rd International Symposium of Microarchitecture (MICRO)*, Atlanta, GA, December 2010.

[9] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 133–146, Big Sky, MT, October 2009.

[10] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*, 3rd Ed., O'Reilly, 2005.

[11] B. Dieny, R. Sousa, G. Prenat, and U. Ebels, Spin-dependent phenomena and their implementation in spintronic devices. In *International Symposium on VLSI Technology, Systems and Applications (VLSI-TSA)*, 2008.

[12] Annie Foong, Bryan Veal, and Frank Hady. Towards SSD-ready enterprise platforms. In *Proceedings of the 1st International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, Singapore, September 2010.

[13] William Gropp, Ewing Lusk, Nathan Doss and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22:789-828, September 1996.

[14] S. Parkin. Racetrack memory: A storage class memory based on current controlled magnetic domain wall motion. In *Device Research Conference (DRC)*, pages 3-6, 2009.

[15] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using Phase-Change Memory technology. In *Proceedings of the 36th International Symposium of Computer Architecture (ISCA)*, Austin, TX, June 2009.

[16] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52:465-480, 2008.

[17] Dongjun Shin. SSD. In *Linux Storage and Filesystem Workshop*, San Jose, CA, February 2008.

[18] David Sitsky and Kenichi Hayashi. An MPI library which uses polling, interrupts and remote copying for the Fujitsu AP1000+. In *Proceedings of the 2nd International Symposium on Parallel Architectures, Algorithms, and Networks (ISPAN)*, Beijing, China, June 1996.

[19] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *Nature*, 453(7191):80-83, May 2008.

[20] Sayantan Sur, Hyun-Wook Jin, Lei Chai, and Dhabaleswar K. Panda. RDMA read based rendezvous protocol for MPI over InfiniBand: design alternatives and benefits. In *Proceedings of the 11th Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 32-39, New York, NY, March 2006.

[21] Xiaojian Wu and Narasimha Reddy. SCMFS: A file system for storage class memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC11)*, Seattle, WA, November 2011.

* Other names and brands may be claimed as the property of others.

Characteristics of Backup Workloads in Production Systems

Grant Wallace   Fred Douglis   Hangwei Qian*   Philip Shilane   Stephen Smaldone
Mark Chamness   Windsor Hsu

Backup Recovery Systems Division
EMC Corporation

Abstract

Data-protection class workloads, including backup and long-term retention of data, have seen a strong industry shift from tape-based platforms to disk-based systems. But the latter are traditionally designed to serve as primary storage and there has been little published analysis of the characteristics of backup workloads as they relate to the design of disk-based systems. In this paper, we present a comprehensive characterization of backup workloads by analyzing statistics and content metadata collected from a large set of EMC Data Domain backup systems in production use. This analysis is both broad (encompassing statistics from over 10,000 systems) and deep (using detailed metadata traces from several production systems storing almost 700TB of backup data). We compare these systems to a detailed study of Microsoft primary storage systems [22], showing that backup storage differs significantly from their primary storage workload in the amount of data churn and capacity requirements as well as the amount of redundancy within the data. These properties bring unique challenges and opportunities when designing a disk-based filesystem for backup workloads, which we explore in more detail using the metadata traces. In particular, the need to handle high churn while leveraging high data redundancy is considered by looking at deduplication unit size and caching efficiency.

1 Introduction

Characterizing and understanding filesystem content and workloads is imperative for the design and implementation of effective storage systems. There have been numerous studies over the past 30 years of file system characteristics for general-purpose applications [1, 2, 3, 9, 15, 20, 22, 26, 30, 31], but there has been little in the way of corresponding studies for backup systems.

Data backups are used to protect primary data. They might typically consist of a full copy of the primary data once per week (i.e., a weekly full), plus a daily backup of the files modified since the previous backup (i.e., a daily incremental). Historically, backup data has been written to tape in order to leverage tape's low cost per gigabyte and allow easy transportation off site for disaster recovery. In the late 1990s, virtual tape (or "VTL") was introduced, which used hard disk storage to mimic a tape library. This allowed for consolidation of storage and faster restore times. Beginning in the early 2000s, deduplicating storage systems [10, 34] were developed, which removed data redundancy and extended the benefits of disk-based backup storage by lowering the cost of storage and making it more efficient to copy data off-site over a network for disaster recovery (replication).

The transition from tape to VTL and deduplicating disk-based storage has seen a strong adoption by the industry. In 2010 purpose-built backup appliances protected 468PB and are projected to protect 8EB by 2015, by which time this will represent a $3.5B market [16]. This trend has made a detailed study of backup filesystem characteristics pertinent for system designers if not long overdue.

In this paper we first analyze statistics from a broad set of 10,000+ production EMC Data Domain systems [12]. We also collect and analyze content-level snapshots of systems that, in aggregate, are to our knowledge at least an order of magnitude larger than anything previously reported. Our statistical analysis considers information such as file age, size, counts, deduplication effectiveness, compressibility, and other metrics. Comparing this to Meyer and Bolosky's analysis of a large collection of systems in Microsoft Corp. [22], we see that backup workloads tend to have shorter-lived and larger files than primary storage. This is indicative of higher data churn rates, a measure of the percentage of storage capacity that is written and deleted per time interval (e.g., weekly), as well as more data sequentiality. These have implications for the design requirements for purpose-built backup systems.

*Current affiliation: Case Western Reserve University.

While summary statistics are useful for analyzing overall trends, we need more detailed information to consider topics such as performance analysis (e.g., cache hit rates) or assessing the effect of changes to system configurations (e.g., varying the unit of deduplication). We address this with our second experimental methodology, using simulation from snapshots representing content stored on a number of individual systems. The content metadata includes detailed information about individual file content, but not the content itself. For example, deduplicating systems will break files into a series of chunks with each chunk represented by a strong hash, sometimes referred to as a fingerprint. We collect the lists of chunk fingerprints and chunk sizes that represent each file as well as the physical layout of these chunks on disk. These collections represent almost 700TB of data and span various data types including databases, emails, workstation data, source code, and corporate application data. These allow us to analyze the stream or file-wise behavior of backup workloads. This type of information is particularly helpful in analyzing the effectiveness of deduplication parameters and caching algorithms.

This study confirms and highlights the different requirements between backup and primary storage. Whereas primary storage capacities have grown rapidly (the total amount of digital data more than doubles every two years [13]), write throughput requirements have not needed to scale as quickly because only a small percentage of the storage capacity is written every week and most of the bytes are longer lived. Contrast this with the throughput requirements of backup systems which, for weekly full backups, must ingest the entire primary capacity every week. The implication is that backup filesystems have had to scale their throughput to meet storage growth. Meeting these demands is a real challenge, and this analysis sheds light on how deduplication and efficient caching can help meet that demand.

To summarize our contributions, this paper:

- analyzes more than 10,000 production backup systems and reports distributions of key metrics such as deduplication, contents, and rate of change;
- extensively compares backup storage systems to a similar study of primary storage systems; and
- uses a novel technique for extrapolating deduplication rates across a range of possible sizes.

The remainder of this paper is organized into the following sections: §2 background and related work, §3 data collection and analysis techniques, §4 analysis of broad trends across thousands of production systems, §5 exploring design alternatives using detailed metadata traces of production systems, and §6 conclusions and implications for backup-specific filesystem design.

2 Background and Related Work

We divide background into three areas: backups (§2.1), deduplication (§2.2), and data characterization (§2.3).

2.1 Backups

Backup storage workloads are tied to the applications which generate them, such as EMC NetWorker or Symantec NetBackup. These backup software solutions aggregate data from online file systems and copy them to a backup storage device such as tape or a (deduplicating) disk-based storage system [7, 34]. As a result, individual files are typically combined into large units, representing for example all files backed up on a given night; these aggregates resemble UNIX "tar" files. Many other types of backup also exist, such as application-specific database backups. Backups usually run regularly, with the most common paradigm being weekly "full" backups and daily "incremental" backups. When files are modified, the incremental backups may have large portions in common with earlier versions, and full backups are likely to have many of their comprising files completely unmodified, so the same data gets written to the backup device again and again.

2.2 Deduplication and other Data Reduction

In backup storage workloads the inherent high degree of data redundancy and need for high throughput make deduplicating techniques important. Deduplication can be performed at the granularity of entire files (e.g., Windows 2000 [5]), fixed blocks (e.g., Venti [29]), or variable-sized "chunks" based on content (e.g., LBFS [24]). In each case, a strong hash (such as SHA-1) of the content, i.e., its "fingerprint," serves as a unique identifier. Fingerprints are used to index content already stored on the system and eliminate duplicate writes of the same data. Because content-defined chunks prevent small changes in content from resulting in unique chunks throughout the remainder of a file, and they are used in the backup appliances we have analyzed, we assume this model for the remainder of this paper. Backup data can be divided into content-defined chunks on the backup storage server, on the backup software intermediary (e.g., a NetBackup server), or on the systems storing the original data. If chunked prior to transmission over a network, the fingerprints of the chunks can first be sent to the destination, where they are used avoid transferring those chunks already present [11, 24].

Traditional compression, such as *gzip*, complements data deduplication. We refer to such compression as "local" compression to distinguish it from compression obtained from identifying multiple copies of data, i.e., deduplication. The systems under study perform local compression after deduplication, combining unique chunks into "compression regions" that are compressed together to improve overall data reduction.

There is a large body of research and commercial efforts on optimizing [14, 21, 34], scaling [8, 10], and improving the storage efficiency [17] of deduplicating backup systems. Our efforts here are mostly complementary to that work, as we are characterizing backup workloads rather than designing a new storage architecture. The impact of the chunk size has been explored in several studies [17, 18, 22, 28], as has delta-encoding of content-defined chunks [18]. However, our study of varying chunk sizes (§5.1) uses real-world workloads that are substantially larger than those used in previous studies. We also develop a novel technique for using a single chunk size to extrapolate deduplication at larger chunk sizes. This is different from the methodology of Kruus, et al. [17], which decides on the fly what chunk size to use at a particular point in a data stream, using the actual content of the stream. Here, we use just the fingerprints and sizes of chunks to form new "merged chunks" at a coarser granularity. We evaluate the effectiveness of this approach by comparing metrics from the approximated merged chunks and native chunking at different sizes, then evaluate the effectiveness of chunking various large-scale datasets over a range of target chunk sizes.

2.3 Data Characterization

The closest work to ours in topic, if not depth, is Park and Lilja's backup deduplication characterization study [27]. It uses a small number of truncated backup traces, 25GB each, to evaluate metrics such as rate of change and compression ratios. By comparison, our paper considers a larger set of substantially larger traces from production environments and aims at identifying filesystem trends related to backup storage.

There have been many studies of primary storage characteristics [1, 2, 3, 9, 15, 20, 22, 26, 30, 31], which have looked at file characteristics, access patterns and caching behavior for primary workloads. Our study measures similar characteristics but for backup workloads. It is interesting to compare the different characteristics between backup and primary storage systems (see §4). For comparison data points we use the most recent study from Microsoft [22], which contains a series of large-scale studies of workstation filesystems. There are some differences that arise from the difference in usage (backup versus day-to-day usage) and some that arise from the way the files are accessed (aggregates of many files versus individual files). For example, the ability to deduplicate whole files may be useful for primary storage [5] but is not applicable to a backup environment in which one file is the concatenation of terabytes of individual files.

3 Data Collection and Analysis Techniques

In conducting a study of file-system data, the most encompassing approach would be to take snapshots of all the systems' data and archive them for evaluation and analysis. This type of exercise would permit numerous forms of interesting analysis including changes to system parameters such as average chunk size and tracking filesystem variations over time.

Unfortunately, full-content snapshots are infeasible for several reasons, the primary one being the need to maintain data confidentiality and privacy. In addition, large datasets (hundreds of terabytes in size each) become infeasible to work with because of the long time to copy and process and the large capacity required to store them. The most practical way of conducting a large-scale study is to instead collect filesystem-level statistics and content metadata (i.e., data about the data).

For this study we collect and analyze two classes of data with the primary aim of characterizing backup workloads to help design better protection storage systems. The first class of data is *autosupport* reports from production systems. Customers can choose to configure their systems to automatically generate and send autosupports, which contain system monitoring and diagnostic information. For our analysis, we extract aggregate information from the autosupports such as file statistics, system capacity, total bytes stored, and others.

The second type of information collected is detailed information about data contained on specific production systems. These collections contain chunk-level metadata such as chunk hash identifiers (fingerprints), sizes, and location on disk. Because of the effort and storage needed for the second type of collection, they are obtained from only a limited set of systems.

The two sets of data complement each other: the autosupports (§3.1) are limited in detail but wide in deployment, while the content metadata snapshots (§3.2) contain great detail but are limited in deployment.

3.1 Collecting Autosupports

The Data Domain systems that are the subject of this study send system data back to EMC periodically, usually on a daily basis. These autosupport reports contain diagnostic and general system information that help the support team monitor and detect potential problems with deployed systems [6]. Over 10,000 of these reports are received per day, which makes them valuable in understanding the broad characteristics of protection storage workloads. They include information about storage usage, compression, file counts and ages, caching statistics and other metrics. Among other things, they can help us understand the distribution of deduplication rates, capacity usage, churn and file-level statistics.

For our analysis, we chose autosupports from a one-week period. From the set of autosupports, we exclude some systems based on certain validation criteria: a system must have been in service more than 3 months and have more than 2.5% of its capacity used. The remaining set consists of more than 10,000 systems with system ages ranging from 3 months to 7 years and gives a broad view of the usage characteristics of backup systems.

We consider these statistics in the aggregate; there is no way to subdivide the 10,000 systems by content type, backup software, or other similar characteristics. In addition, we must acknowledge some possible bias in the results. This is a study of EMC Data Domain customers who voluntarily provide autosupport data (the vast majority of them do); these customers tend to use the most common brands of backup software and typically have medium to large computing environments to protect.

3.2 Collecting Content Metadata

In this study, we work with deduplicated stores which enable us to collect content metadata more efficiently. On deduplicated systems a chunk may be referenced many times, but the detailed information about the chunk need be stored just once. Figure 1 shows a schematic of a deduplicated store. We collect the file recipes (listing of chunk fingerprints) for each file and then collect the deduplicated chunk metadata from the storage containers, as well as sub-chunk fingerprints (labeled "sub-fps") as described below. The file recipe and per-chunk metadata can be later combined to create a per-file "trace" comprised of a list of detailed chunk statistics as depicted on the bottom right of the figure. (Note that this "trace" is not a sequence of I/O operations but rather a sequence of file chunk references that have been written to a backup appliance, from oldest to newest.) Details about the trace, including its efficient generation, are described in §3.2.3.

In this way, the collection time and storage needed for the trace data is proportional to the deduplicated size. This can lead to almost a 10X saving for a typical backup storage system with 10X deduplication. In addition, some of the data analysis can be done on the deduplicated chunk data. This type of efficiency becomes very important when dealing with underlying datasets of hundreds of terabytes in size. These systems will have tens of billions of chunks and even the traces will be hundreds of gigabytes in size.

3.2.1 Content Fields Collected

For the content metadata snapshots, we collect the following information (additional data are not discussed due to space limitations):

- Per-chunk information such as size, type, SHA-1 hash, subchunk sizes and abbreviated hashes.

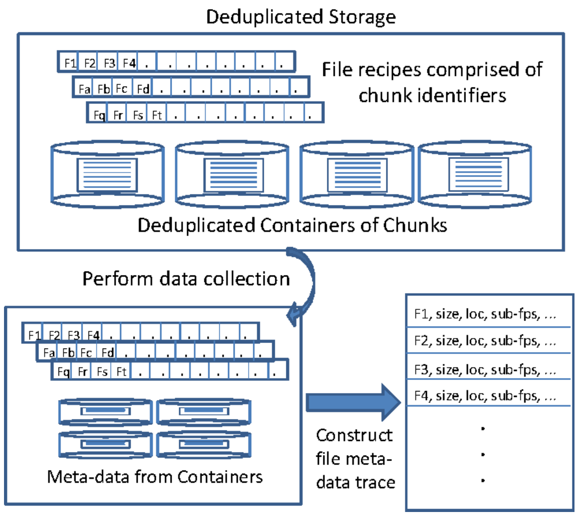

Deduplicated Storage
F1 F2 F3 F4
Fa Fb Fc Fd
Fq Fr Fs Ft
File recipes comprised of chunk identifiers
Deduplicated Containers of Chunks
Perform data collection
Meta-data from Containers
Construct file meta-data trace
F1, size, loc, sub-fps, ...
F2, size, loc, sub-fps, ...
F3, size, loc, sub-fps, ...
F4, size, loc, sub-fps, ...

Figure 1: Diagram of Data Collection Process

- Per-file information such as file sizes, modification times, and fingerprints of each chunk in the file.
- Disk layout information such as location and grouping of chunks on disk.

One of the main goals for these collections was to look at throughput and compression characteristics with different system configurations. The systems studied were already chunked at 8KB on average with the corresponding SHA-1 hash values available. We chose to sub-chunk each 8KB chunk to, on average, 1KB and collected abbreviated SHA-1 hashes for each 1KB sub-chunk. Sub-chunking allowed us to investigate deduplication rates at various chunk sizes smaller than the default 8KB, as described in §5.1.

3.2.2 Creating Traces from Metadata Snapshots

The collected content metadata can be used to create per-file traces of chunk references. These traces are the ordered list of chunk metadata that comprise a file. For example, the simplest file trace would contain a file-ordered list of the chunk fingerprints and sizes that comprise the file. More detailed traces might also include other per-chunk information such as disk location. These file traces can be run through a simulator or analyzed in other ways for metrics such as deduplication or caching efficiency.

The per-file traces can be concatenated together, for example by file modification time (mtime), to create a representative trace for the entire dataset. This can be used to simulate reading or writing all or part of the system contents; our analyses in §5 are based on such traces.

For example, to simulate a write workload onto a new system, we could examine the sequence of fingerprints in order and pack new (non-duplicate) chunks together into storage containers. The storage containers could then be used as the unit of caching for later fingerprints in the sequence [34]. This layout represents a pristine storage system, but in reality, chunk locality is often fragmented because garbage collection of deleted files causes live chunks from different containers to be merged into new containers. Instead of using the pristine layout, we could use the container layout of chunks as provided by the metadata snapshot from the production system, which gives a more realistic caching analysis.

To simulate a read workload, we would examine the sequence of fingerprints in order and measure cache efficiency by analyzing how many container or compression-region loads are required to satisfy the read. Compression-regions are the minimal unit of read, since the group of chunks have to be uncompressed together, but reading whole containers may improve efficiency. While reading the entire dataset trace would be a complete restore of all backups, perhaps more realistically, only the most recent full backup should be read to simulate a restore.

To approximate the read or write of one full backup of the trace requires knowledge of what files correspond to a backup. Since we don't have the backup file catalog, we are not able to determine a full backup at file granularity. Instead we divide the trace into a number of equal sized intervals, with the interval size based on the deduplication rate. For instance, if the deduplication rate is 10X then we estimate that there are about 10 full backups on the system, i.e., the original plus 9 identical copies. In this example we would break the trace into 10 intervals approximating about one backup per interval. This is an approximation: in practice, the subsequent backups after the first will not be identical but will have some data change. But this is a reasonable approach for breaking the caching analysis into intervals, which allows for warming the cache and working on an estimated most-recent backup copy.

3.2.3 Efficient Analysis of Filesystem Metadata

The file trace data collected could be quite large, sometimes more than a terabyte in size, and analyzing these large collections efficiently is a challenge. Often the most efficient way to process the information is by use of out-of-core sorting. For instance, to calculate deduplication ratios we sort by fingerprint so that repeated chunks are adjacent, which then allows a single scan to calculate the unique chunk count. As another example, to calculate caching effectiveness we need to associate fingerprints with their location on disk. We first sort by fingerprint and assign the disk location of the first instance to all duplicates, then re-sort by file mtime and offset to have a time-ordered trace of chunks, with container locations, to evaluate.

Even the process of merging file recipes with their associated chunk metadata to create a file trace would be prohibitively slow without sorting. We initially implemented this merge in a streaming fashion, looking up chunk locations and pre-fetching neighboring chunks into a cache, much as an actual deduplication system would handle a read. But the process was slow because of the index lookups and random seeks on an engineering workstation with a single disk. Eventually we switched this process to also use out-of-core sorting. We use a four-step process of (1) sorting the file recipes by fingerprint, (2) sorting the chunk metadata collection by fingerprint, (3) merging the two sets of records, and (4) sorting the final record list by logical position within the file. This generates a sequence of chunks ordered by position within the file, including all associated metadata.

4 Trends Across Backup Storage Systems

We have analyzed the autosupport information from more than 10,000 production deduplicated filesystems, taken from an arbitrary week, July 24, 2011. We compare these results with published primary storage workloads from Microsoft Corp. [22]. The authors of that study shared their data with us, which allows us to graph their primary workload results alongside our backup storage results. The Microsoft study looked at workstation filesystem characteristics for several different time periods; we compare to their latest, a one month period in 2009 which aggregates across 857 workstations.

Backup storage file characteristics are significantly different from the Microsoft primary workload. Data-protection systems have generally larger, fewer and shorter lived files. This is an indication of more churn within the system but also implies more data sequentiality. The following subsections detail some of these differences. In general, figures present both a histogram (probability distribution) and a cumulative distribution function (CDF), and when counts are presented they are grouped into buckets representing ranges, on a log scale, with labels centered under representative buckets.

4.1 File Size

A distinguishing characteristic between primary and backup workloads is file size. Figure 2 shows the file size distribution, weighted by bytes contained in the files, for both primary and backup filesystems. For backup this size distribution is about 3 orders of magnitude larger than for primary files. This is almost certainly the result of backup software combining individual files together from the primary storage system into "tar-like" collections. Larger files reduce the likelihood of whole-file deduplication but increase the stream locality within the system. Notice that for backup files a large percentage of the space is used by files hundreds of gigabytes in

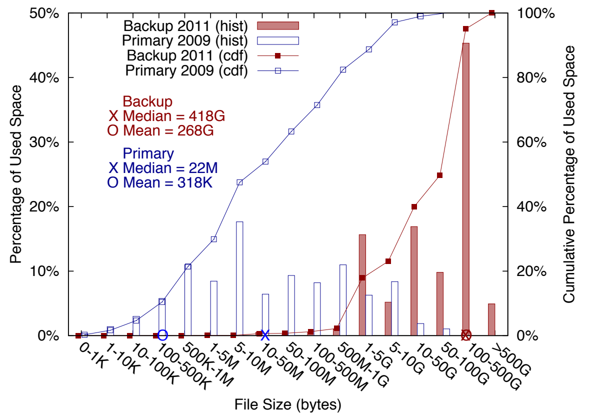


Figure 2: File size

size. Small file optimizations that make sense for primary storage such as embedding data in inodes or use of disk block fragments may not make sense for backup filesystems where large allocation units can provide more efficient metadata use.

4.2 File and Directory Count

File and directory counts are typically much lower in backup workloads. Similar to the effect of large file sizes, having a low file count (Figure 3(a)) results from having larger tar-type concatenations of protected files. The low directory count (Figure 3(b)) is a result of backup applications using catalogs to locate files. This is different from typical user-organized filesystems where a directory hierarchy is used to help order and find data. Looking at the ratio of file to directory count (Figure 3(c)), we can see again that backup workloads tend to use a relatively flat hierarchy with several orders of magnitude more files per directory.

4.3 File Age

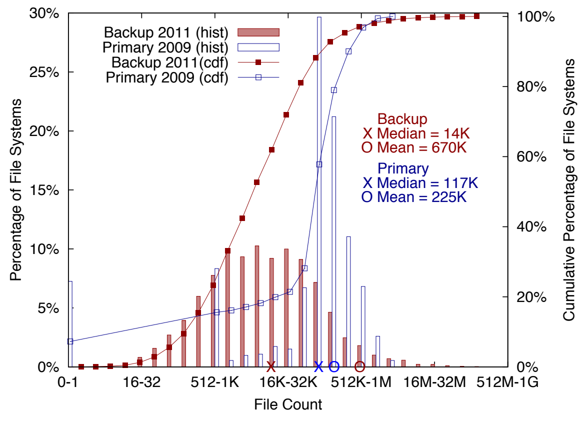
Figure 4 shows the distribution of file ages weighted by their size. For backup workloads the median age is about 3 weeks. This would correspond to about 1/2 the retention period, implying data retention of about 6 weeks. Short retention periods lead to higher data churn, as seen next.

4.4 Filesystem Churn

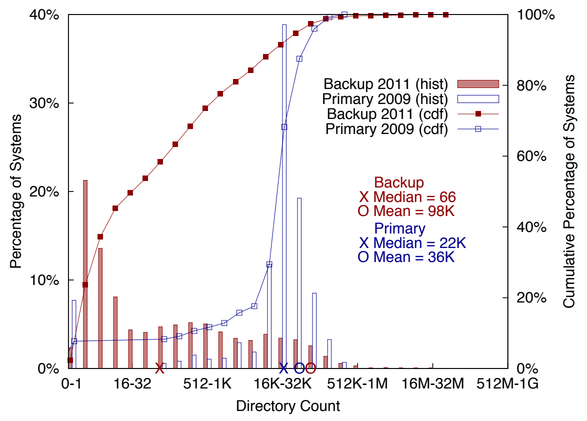
Filesystem churn is a measure of the percentage of storage that is freed and then written per time period, for instance in a week. Figure 5 shows a histogram of the weekly churn occurring across the studied backup systems.

On average about 21% of the total stored data is freed and written per week. This high churn rate is driven by backup retention periods. If a backup system has a 10-week retention policy, about 10% of the data needs to be written and deleted every week. The median churn rate is about 17%, corresponding to almost a 6-week retention period, which correlates well with the median byte age of about 3 weeks.

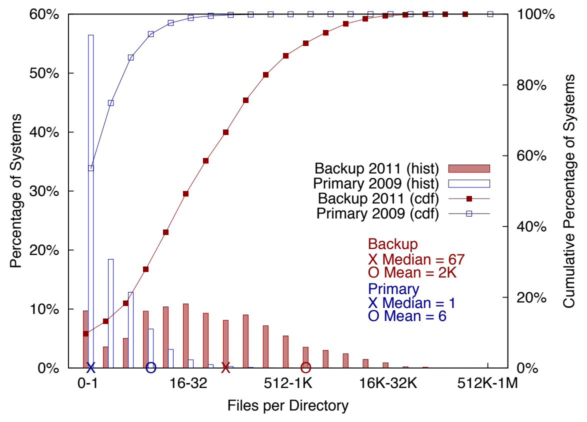
This has implications for backup filesystems: such filesystems must be able not only to write but also re-



(a) File count



(b) Directory count



(c) Files per directory

Figure 3: File and directory counts

claim large amounts of space on a weekly basis. Storage technologies with limited erase cycles, such as flash memory, may not be a good fit for this workload without care to avoid arbitrary overwrites from file system cleaning [23].

To some extent, deduplicating storage systems help alleviate this problem because less physical data needs to be cleaned and written each week. The ratio of data written per week to total stored is similar whether those are calculated from pre-deduplication file size or post-deduplicated storage size; this is expected as long as the deduplication ratio is relatively constant over time.

Note also that backup churn rates increase quickly over time. They follow the same growth rate as the underlying primary data, (i.e., doubling every two years). To meet the high ingest rates, backup filesystems can leverage the high data redundancy of backup workloads. In-line deduplication of file streams can eliminate many of the disk writes and increase throughput. Doing so effectively requires efficient caching, which is studied further in §5.

4.5 Read vs Write Workload

Data-protection systems are predominately write workloads but do require sufficient read bandwidth in order to stream the full backup to tape, replicate changed data offsite, and provide for timely restores. Figure 6 shows the distribution of the ratio of bytes written vs total I/O bytes, excluding replication and garbage collection. About 50% of systems have overwhelmingly more writes than reads (90%+ write). Only about 20% of systems have more reads than writes.

These I/O numbers underestimate read activity because they do not include reads for replication. However, since during replication an equal number of bytes are read by the source as written by the destination, the inclusion of these statistics might change the overall percentages but not change the conclusion that writes predominate.

This is the opposite of systems with longer-lived bytes such as primary workloads, which typically have twice as many reads as writes [20]. The high write workloads are again indicative of high-churn systems where large percentages of the data are written every week.

4.6 Replication

For disaster recovery, backup data is typically replicated off-site to guard against site-level disasters such as fires or earthquakes. About 80% of the production systems replicate at least part of their backup data each week.

Of the systems that replicate, Figure 7 shows the percentage of bytes written in the last 7 days that are also replicated (either to or from the system). On average almost 100% of the data is replicated on these systems.

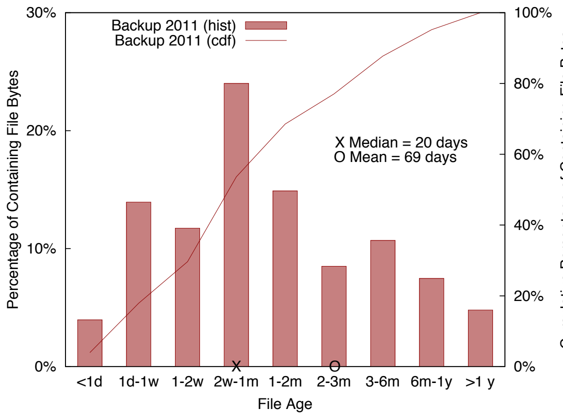


Figure 4: File age

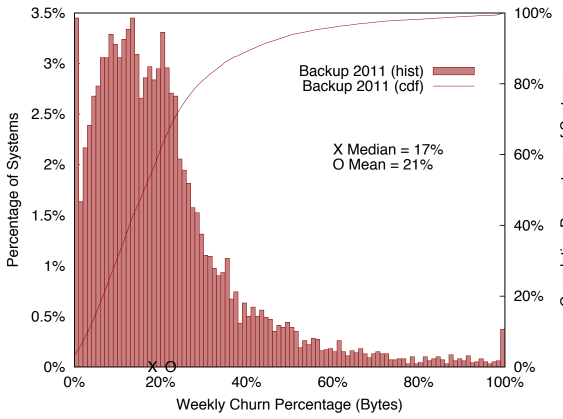


Figure 5: Weekly churn

Notice that some systems replicate more data than was written in this time period. This can be due to several causes: some systems replicate to more than one destination and some systems perform cascaded replication (they receive replicated data and in turn replicate it to another system).

The high percentage of replicated data increases the need for read throughput, resulting in a slightly more balanced read to write ratio than one might expect from just backup operations (write once, read rarely). This implies that while backup systems must provide excellent write performance, they cannot ignore the importance of read performance.

In concurrent work, cache locality for delta compression is analyzed in the context of replication, including information from production autosupport results [32].

4.7 Capacity Utilization

Figure 8 shows the distribution of storage usage for both primary and backup systems. Backup systems skew toward being more full than primary systems, with the

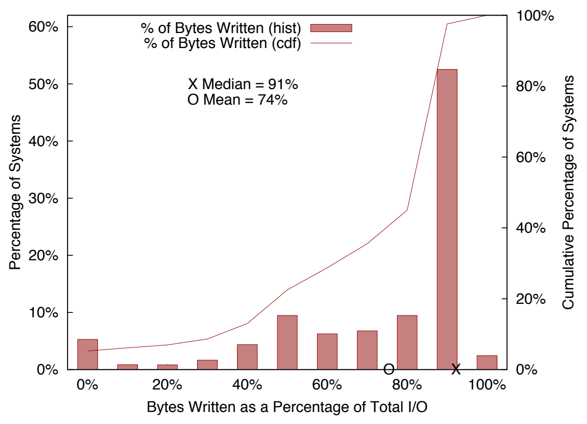


Figure 6: Read/write byte ratio

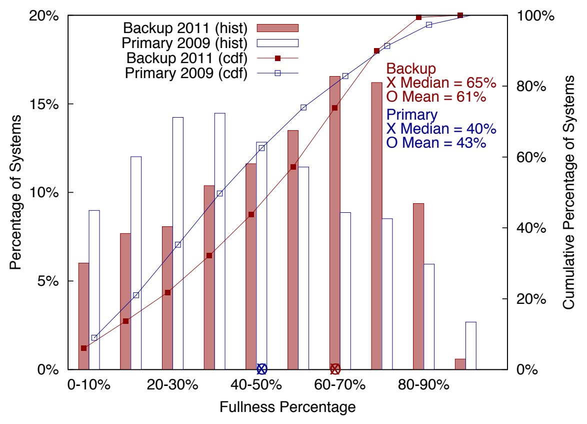


Figure 8: Fullness

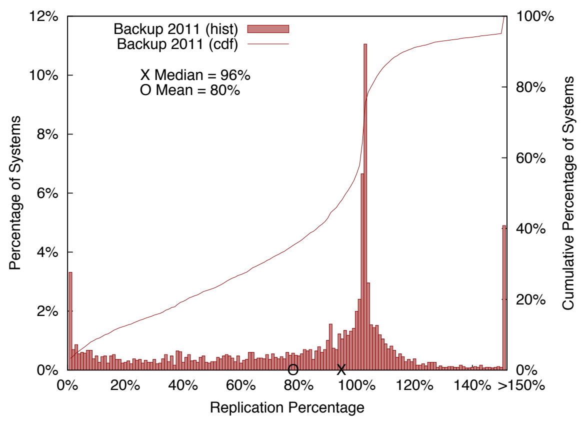


Figure 7: Percent of data replicated

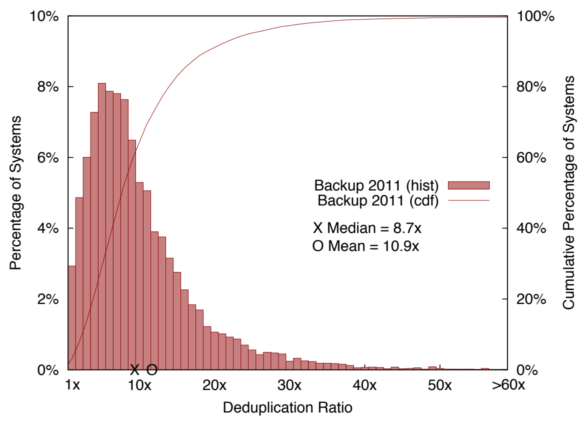


Figure 9: Deduplication

backup system modal (most frequent) utilization about 60–70% full. In contrast primary systems are about 30–40% full. The gap in mean utilization may reflect differences in the goals of the administrators of the two types of systems: while performance and capacity are both important in each environment, there is a greater emphasis in data protection on dollar-efficient storage, while primary storage administrators may stress performance. Also, backup administrators have more flexibility in balancing the data protection workloads across systems, as they can shorten retention periods or reduce the domain of protected data. Achieving higher utilization helps to optimize the cost of overall backup storage [6].

4.8 Deduplication Rates

The amount of data redundancy is one of the key characteristics of filesystem workloads and can be a key driver of cost efficiency in today's storage systems. Here we compare the deduplication rates of backup filesystem workloads with those of primary storage as reported by Meyer and Bolosky [22]. Figure 9 indicates that deduplication rates for backup storage span a wide range across system and workloads with a mean of 10.9x. This is dramatically higher than for primary workloads with a mean of about 3x in the Microsoft workload. The main difference is that backup workloads generally retain multiple copies of data.

Additionally, backups are usually contained within large tar-type archives that do not lend themselves to whole-file deduplication. When these larger files are subdivided into chunks for deduplication, the chunk size can have widely varying effects on deduplication effectiveness (see §5.1).

4.8.1 Compression

Data Domain systems aggregate new unique chunks into compression regions, which are compressed as a single unit (approximately 128KB before compression). Since there is usually spatial locality between chunks that are written together, the compressibility of the full region is much greater than what might be achieved by compressing each 8KB chunk in isolation.

#	Snapshot	Class	Data Type	Size (TB)	Dedup. Ratio	1-Wk Dedup.	MedAge (Weeks)	Retention Time	Update Freq.
1	homedirs	LT-B	Home Directories	201	14.0	3.0	3.49	1–3 years	MF
								5 weeks	DF
2	database1	B	Database	177	5.1	2.7	2.21	1 month	MF/DI
3	email	B	Email	146	9.6	1.1	1.36	15 days	DF
4	fileservers	B	Windows Fileservers	60	5.9	1.7	5.80	3 months	WF/DI
5	mixed1	B	Mixed DB/Email/User	47	6.0	2.4	3.24	1–3 months	MF/DI
6	mixed2	B	Workstations, Servers	43	11.0	3.0	9.44	4–6 months	WF/DI
7	workstations	B	Workstations	4.5	7.5	2.3	13.56	4 months	WF/DI
8	database2	B	Database	3.8	2.2	1.3	0.23	3 days	DF

Table 1: Collected Datasets. Class can be **B** "Backup," **LT-B** or"Long Term Backup." Retention can be **MF** "Monthly Full," **WF** "Weekly Full," **DF** "Daily Full," or **DI** "Daily Incremental." We report cumulative and one-week deduplication. **MedAge** is the mid-point at which half the data is newer or older.

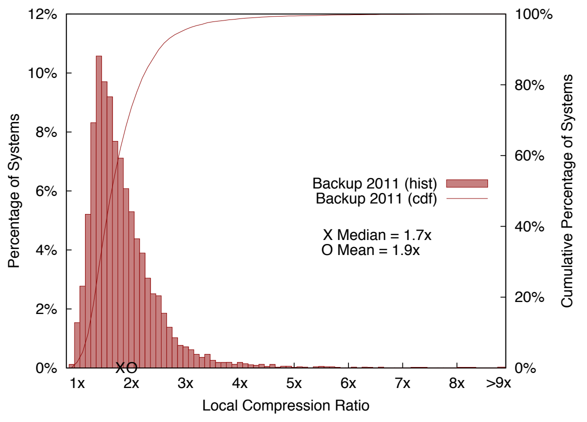


Figure 10: Local compression

As a result, the effectiveness of post-deduplication compression in a backup workload will typically be comparable to that of a primary workload. Figure 10 shows the local compression we see across production backup workloads, with a mean value of almost 2X as the expected rule of thumb [19]. But as can be seen, there is also a large variation across systems with some data inherently more random than others.

5 Sensitivity Analyses of Deduplicating Backup Systems

Deduplication has enabled the transition from tape to disk-based data protection. Storing multiple protected copies on disk is only cost effective when efficient removal of data redundancy is possible. In addition deduplication provides for higher write throughput (fewer disk writes), which is necessary to meet the high churn associated with backup storage (see §4.4). However, read performance can be negatively impacted by the fragmentation introduced by deduplication [25].

In this section we use trace-driven simulation to evaluate the effect of chunk size on deduplication rates (§5.1) and to evaluate alternatives for caching the fingerprints used for detecting duplicates (§5.2). First we describe the metadata collections, which are used for the sensitivity analyses, in greater detail. Table 1 lists the data sets collected and their properties, in decreasing order of pre-deduplication size. They span a wide range of sizes and deduplication rates. Most are straightforward "backup" workloads while one includes some data meant for long-term retention. They range from traces representing as little as 4–5TB of pre-deduplicated content up to 200TB. The deduplication ratio (using the default 8KB target chunk size) also has a large range, from as little as 2.2 to as much as 14.0; the data sets with the lowest deduplication have relatively few full backups, with the extreme case being a mere 3 days worth of daily full backups.

Deduplication over a prolonged period can be substantial if many backups are retained, but how much deduplication is present over smaller windows, and how skewed is the stored data? These metrics are represented in the **1-Wk Dedup.** and **MedAge** columns. The former estimates the average deduplication seen within a single week, which typically includes a full backup plus incrementals. This is an approximation of the intra-full deduplication which cannot be determined directly because the collected datasets do not provide information about full backup file boundaries. The median age is the point by which half the stored data was first written, and it provides a view into the retention and possible deduplication. For instance, half of the data in `homedirs` had been stored for 3.5 weeks or less. With daily full backups stored 5 weeks we would expect a median age of 2.5 weeks, but the monthly full backups compensate and increase the median.

5.1 Effect of Varying Chunk Size

The systems studied use a default average chunk size of 8KB, but smaller or larger chunks are possible. Varying the unit of deduplication has been explored many times in the past, usually by chunking the data at multiple sizes and comparing the deduplication achieved [18, 22, 28]; it is also possible to vary the deduplication unit dynamically [4, 17]. The smaller the average chunk size, the finer-grained the deduplication. When there are long regions of unchanged data, the smaller chunk size has little effect, since any chunk size will deduplicate equally well. When there are frequent changes, spaced closer together than a chunk, all chunks will be different and fail to deduplicate. But when the changes are sporadic relative to a given chunk size, having smaller chunks can help to isolate the parts that have changed from the parts that have not.

5.1.1 Metadata Overhead

Since every chunk requires certain metadata to track its location, the aggregate overhead scales inversely with the chunk size. We assume a small fixed cost, 30 bytes, per physical chunk stored in the system and the same cost per logical chunk in a file recipe (where physical and logical are post-deduplication and pre-deduplication, respectively). The 30 bytes represent the cost of a fingerprint, chunk length, and a small overhead for other metadata.

Kruus, et al., described an approach to chunking data at multiple granularities and then selecting the most appropriate size for a region of data based on its deduplication rate [17]. They reported a reduction in deduplication effectiveness by a factor of $\frac{1}{(1+f)}$, where f is defined as the metadata size divided by the average chunk size. For instance, with 8KB chunks and 30 bytes of metadata per chunk, this would reduce the effectiveness of deduplication by 0.4%.

However, metadata increases as a function of both post-deduplication physical chunks and pre-deduplication logical chunks, i.e., it is a function of the deduplication rate itself. If the metadata for the file recipes is stored outside the deduplication system, the formula for the overhead stated above would be correct. If the recipes are part of the overhead, we must account for the marginal costs of each logical chunk, not only the post-deduplication costs. Since the raw deduplication D is the ratio of logical to physical size (i.e., $D = L/P$) while the real deduplication D' includes metadata costs ($D' = \frac{L}{P+fP+fL}$), we can substitute $L = DP$ in the latter equation to get:

$$D' = \frac{D}{1+f(1+D)}.$$

Intuitively, we are discounting the deduplication by the amount of metadata overhead for one copy of the physical data and D copies of the logical data. For a deduplication rate of 10X, using this formula, this overhead would reduce deduplication by 4% rather than 0.4%.

However, as chunks get much smaller, the metadata costs for increasing the number of chunks can dominate the savings from a smaller chunksize. We can calculate the breakeven point at which the net physical space using chunksize C_1 is no greater than using twice that chunksize (C_2, where $C_2 = 2C_1$). First, we note that $f_1 = 2f_2$ since the per-chunk overhead doubles. Then we compare the total space (physical post-deduplication P_i plus overhead) for both chunk sizes, using a single common logical size L:

$$P_1 + 2f(L+P_1) \leq P_2 + f(L+P_2).$$

Since $D_i = L/P_i$ we can solve for the necessary D_1:

$$D_1 \geq \frac{D_2(1+2f)}{1+f(1-D_2)}.$$

This inequality shows where the improvement in raw deduplication (not counting metadata) is at least as much as the increased metadata cost.[1] As an example, with the 30 bytes of overhead and 10X raw deduplication at **2**KB chunks, one would need to improve to 11.9X or more raw deduplication at the **1**KB chunk size to fare at least as well.

5.1.2 Subchunking and Merging Chunks

We are able to take snapshots of fingerprints but not of content, so it is not possible to rechunk content at many sizes. While we could chunk data from a system at numerous sizes at the time the snapshot is created, that would require more processing and more storage than are feasible. Thus, to permit the analysis of pre-chunked data, for which we can later store the fingerprints but not the content, we take a novel approach. To get smaller chunks than the native 8KB size, during data collection we read in a chunk at its original size, *sub*-chunk it at a single smaller size (1KB), and store the fingerprints and sizes of the smaller sub-chunks along with the original chunk metadata. We can then analyze the dataset with 1KB chunks, or merge 1KB chunks into larger chunks

[1]There is also a point at which the deduplication at one size is so high that the overhead from doubling the metadata costs would dominate any possible improvement from better deduplication, around 67X for our 30-byte overhead. Also, the formula applies to a single factor of two but could be adjusted to allow for other chunk sizes.

(such as 2KB or 4KB on average). We can also merge the original 8KB chunks into larger units (powers of two up to 1MB). To keep the merged chunks distinct from the native 8KB chunks or the 1KB sub-chunks, we will refer to merged chunks as *m*chunks.

For a given average chunk size, the system enforces both minimum and maximum sizes. To create an *m*chunk within those constraints, we group a minimum number of chunks (or sub-chunks) to reach the minimum size, then determine how many additional chunks to include in the *m*chunk in a content-aware fashion, similar to how chunks are created in the first place. For instance, to merge 1KB chunks into 4KB *m*chunks (2KB minimum and 6KB maximum), we would start with enough 1KB-average chunks to create at least a 2KB *m*chunk, then look at the fingerprints of the next *N* chunks, where the *N*th chunk considered is the last chunk that, if included in the *m*chunk, would not exceed the maximum chunk size of 6KB.

At this point we have a choice among a few possible chunks at which to separate the current *m*chunk from the next one. We need a content-defined method to select which chunk to use as the breakpoint, similar to the method used for forming chunks in the first place within a size range. Here, we select the chunk with the *highest value fingerprint* as the breakpoint. Since fingerprints are uniformly distributed, and the same data will produce the same fingerprint, this technique produces consistent results (with sizes and deduplication comparable to chunking the original data), as we discuss next. We experimented with several alternative selection methods with similar results.

5.1.3 Evaluation

A key issue in this process is evaluating the error introduced by the constraints imposed by the merging process. We performed two sets of experiments on the sub-chunking and merging. The first was done on full-content datasets, to allow us to quantify the difference between ground truth and reconstructed metadata snapshots. We used two of the datasets from an earlier deduplication study [8], approximately 5TB each, to compute the "ground truth" deduplication and average chunk sizes. We compare these to the deduplication rate and average when merging chunks. (The datasets were labeled "workstations" and "email" in the previous study, but the overall deduplication rates are reported slightly differently because here we include additional overhead for metadata; despite the similar naming, these datasets should not be confused with the collected snapshots in Table 1.) Table 2 shows these results: the average chunk size from merging is consistently about 2–3% lower. For the workstations dataset, the deduplication rate is slightly higher, presumably due to smaller deduplication units, while for the email dataset the deduplication rate is somewhat lower (by 4–7%) with merging than when the dataset is chunked at a given size. However, the numbers are all close enough to serve as approximations to natural chunking.

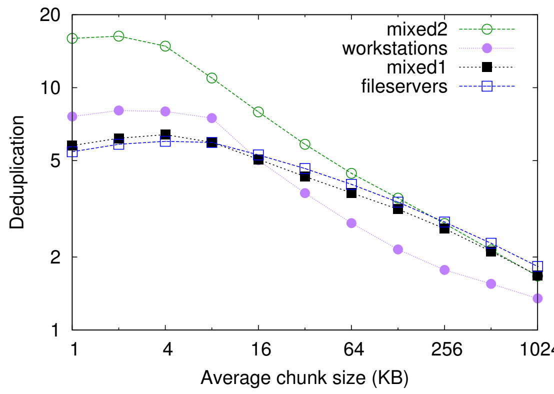


Figure 11: Deduplication obtained as a function of chunk size, using the merging technique. Both axes are on a log scale.

The second set of experiments, shown in Figure 11, was performed on a subset of the collected datasets (we selected a few for clarity and because the trends are so similar). For these we have no "ground truth" other than statistics for the original 8K chunks, but we report the deduplication rates as a function of chunk size as the size ranges from 1K sub-chunks to 1024KB (1MB) *m*chunks. The 1KB sub-chunks are used to merge into 2-4KB *m*chunks and the 8KB original chunks are used for the larger ones.

Looking at both the "ground truth" datasets and the snapshot analyses, we see that deduplication decreases as the chunk size increases, a result consistent with many similar studies. For most of the datasets this is an improvement of 20–40% for each reduction of a power of two, though there is some variability. As mentioned in §5.1.1, there is also a significant metadata overhead for managing smaller chunks. In Figure 11, we see that deduplication is consistently worse with the smallest chunks (1KB) than with 2KB chunks, due to these overheads: at that size the metadata overhead typically reduces deduplication by 10–20%, and in one case nearly a factor of two. Large chunk sizes also degrade deduplication; in fact, the `database1` dataset (not plotted) gets no deduplication at all for large chunks. Excluding metadata costs, the datasets in Table 2 would improve deduplication by about 50% when going from 2KB to 1KB average chunk size, but when these costs are included the improvement is closer to 10%; this is because for those datasets, the improvement in deduplication sufficiently compensates for the added per-chunk metadata.

Target Size (KB)	Workstations-full				Email-full			
	Ground Truth		Merged		Ground Truth		Merged	
	Dedup.	Avg Size (KB)	Dedup.	Avg Size (KB)	Dedup.	Avg Size (KB)	Dedup.	Avg Size (KB)
1	10.48	1.04	*N/A*		10.88	1.05	*N/A*	
2	9.29	2.08	9.30	2.02	9.54	2.09	9.19	2.03
4	7.28	4.15	7.28	4.09	7.84	4.18	7.48	4.08
8	5.49	8.34	*N/A*		6.61	8.33	*N/A*	
16	4.16	16.68	4.17	16.18	5.37	16.64	5.08	16.19
32	3.23	33.33	3.24	32.69	4.18	33.41	3.90	32.66
64	2.59	66.67	2.64	65.62	3.21	66.43	3.03	65.52

Table 2: Comparison of the ground truth deduplication rates and chunk sizes, compared to merging sub-chunks (to 4KB) or chunks (above 8KB), on two full-content snapshots. The target sizes refer to the *desired* average chunk size. The ground truth values for 1KB and 8KB average chunk sizes are included for completeness.

Data and analysis such as this can be useful for assessing the appropriate unit of deduplication when variable chunk sizes are supported [4, 17].

5.2 Cache Performance Analysis

In deduplicating systems, the performance bottleneck is often the lookup for duplicate chunks. Systems with hundreds of terabytes of data will have tens of billions of chunks. With each chunk requiring about 30 bytes of metadata overhead, the full index will be many hundreds of gigabytes. On today's systems, indexes of this size will not fit in memory and thus require an on-disk index, which has high access latency [34].

Effective caching techniques are necessary to alleviate this index lookup bottleneck, and indeed there have been numerous efforts at improving locality (e.g., Data Domain's Segment-Informed Stream Locality [34], HP's sparse indexing [21], and others). These studies have indicated that leveraging stream locality in backup workloads can significantly improve write performance, but their analyses have been limited to a small number of workloads and a fixed cache size. Unlike previous studies, we analyze for both read and write workloads across a broader range of datasets and examine the sensitivity of cache performance to cache sizes and the unit of caching.

5.2.1 Caching Effectiveness for Writes

As seen in §4, writes are a predominant workload for backup storage. Achieving high write throughput requires avoiding expensive disk index lookups by having an effective chunk-hash cache. The simplest caching approach would be to use an LRU cache of chunk hashes. An LRU cache relies on duplicate chunks appearing within a data window that is smaller than the cache size. For backup workloads, duplicate chunks are typically found once per full backup, necessitating a cache sized as large as a full backup per client. This is prohibitively large.

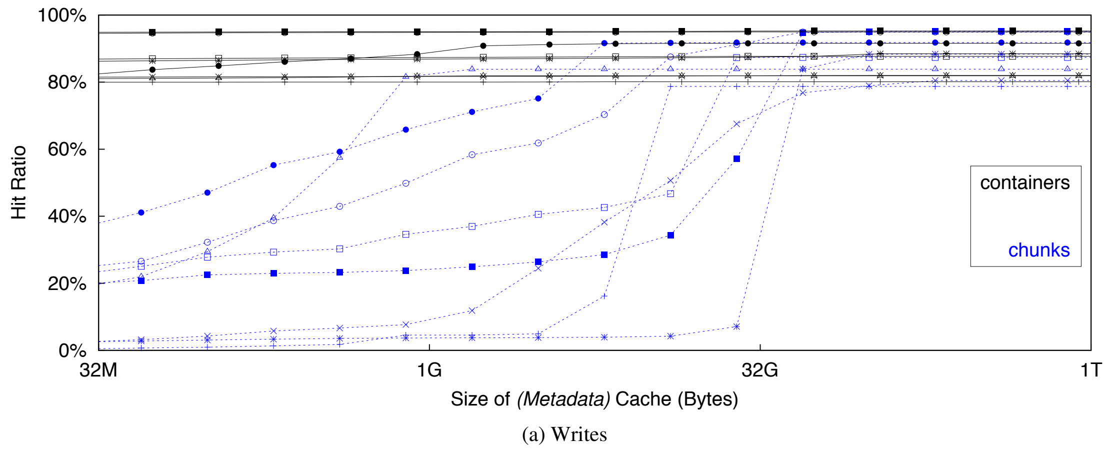
To improve caching efficiency, stream locality hints can be employed. [21, 34]. Files are typically grouped in a similar order for each backup, and re-ordering of intra-file content is rare. The consistent stream-ordering of content can be leveraged to load the hashes of nearby chunks whenever an index lookup occurs. One method of doing so is to pack post-deduplicated chunks from the same stream together into disk regions.

To investigate caching efficiency, we created a cache simulator to compare LRU versus using stream locality hints. The results for writing data are shown in Figure 12(a). The LRU simulator does per-chunk caching and its results are reported in the figure with the dotted blue lines. The stream locality caching groups chunks into 4MB regions called "containers" and its results are reported in that figure with solid black lines. We simulate various cache sizes from 32MB up to 1TB where the cache only holds chunk fingerprints (not the chunk data itself).[2] For these simulations, we replay starting with the beginning of the trace to warm the cache and then record statistics for the final interval representing approximately the most recent backup.

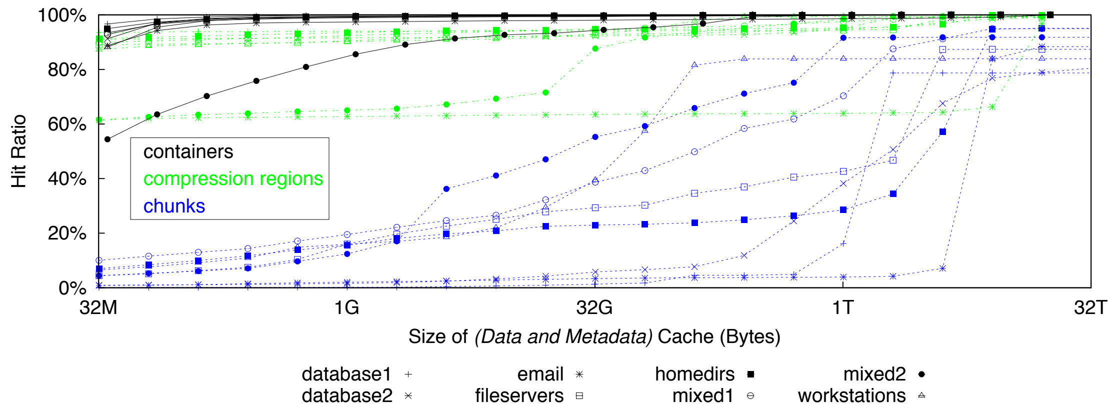
Note that deduplication write workloads have two types of compulsory misses, those when the chunk is in the system but not represented in the cache (duplicate chunks), and those for new chunks that are not in the system (unique chunks). This graph includes both types of compulsory misses. Because the misses for new chunks are included, the maximum hit ratio is the inverse of the deduplication ratio for that backup.

Using locality hints reduces the necessary cache size by up to 3 orders of magnitude. Notice that LRU does achieve some deduplication with a relatively small cache, i.e., 5-40% of duplicates could be identified with a 32MB

[2]To make the simulation tractable, we sampled 1 in 8 cache units, then scaled the memory requirement by the sampling rate. We validated this sampling against unsampled runs using smaller datasets. The cache size is a multiple of the cache unit for a type; therefore, data points of similar cache size do not align completely within Figure 12(a) and (b). We crop the results of Figure 12(a) at 32MB to align with Figure 12(b).



(a) Writes



(b) Reads

Figure 12: Cache results for writing or reading the final portion of each dataset. For writes, the cache consists just of metadata, while for reads it includes the full data as well and must be larger to have the same hit ratio. Differences in marks represent the datasets, while differences in color represent the granularity of caching (containers, chunks, or in the case of reads, compression regions).

cache (dotted blue lines). These duplicates which occur relatively close together in the logical stream may represent incremental backups that write smaller regions of changed data. However, effective caching is not typically achieved with the LRU cache until the cache size is many gigabytes in size, likely representing, at that point, a large portion of the unique chunks in the system. In contrast, using stream locality hints achieves good deduplication hit rates with caches down to 32MB in size (solid black lines across the top of the figure). Since production systems typically handle tens to hundreds of simultaneous write streams, each stream with its own cache, keeping the per-stream cache size in the range of megabytes of memory is important.

5.2.2 Caching Effectiveness for Reads

Read performance is also important in backup systems to provide fast restores of data during disaster recovery. In this subsection, we present a read caching analysis similar to that of the previous subsection.

There are three main differences between the read and write cache analysis. The first is that read caches contain the data whereas the write caches only needs the chunk fingerprints. The second is that reads have only one kind of compulsory miss, those due to cache misses, while writes can also miss due to the first appearance of a chunk. The third is that in addition to analyzing stream locality hints at the container level (which represents 4MB of chunks) we also analyze stream locality at the compression-region level, a 128KB grouping of chunks.

Figure 12(b) shows the comparison of LRU with stream locality hints at the container and compression-region granularity for read streams. The effectiveness of using stream locality hints is even more exaggerated here than for write workloads. Stream locality hints still allow cache sizes of less than 32MB for container level caching (solid black lines), but chunk-level LRU (dotted blue lines) now requires up to several terabytes of cache (chunk data) to achieve effective hit rates. There is now a 4-6 order of magnitude difference in required cache sizes. Compression-region caching (dashed green lines) is as effective as container-level for 6 of the datasets, but 2 show significantly degraded hit ratios. These two datasets are from older systems which apparently have significant fragmentation at the compression-region level, which is smoothed out at the container level.

Fragmentation has two implications on performance. One is that data that appear consecutively in the logical stream can be dispersed physically on disk, impacting read performance [25]. Another is that the unit of transfer may not correspond to the unit of access; e.g., one may read a large unit such as a container just to access a small number of chunks. The impact of fragmentation on performance is the subject of recent and ongoing work (e.g., SORT [33]).

6 Conclusion

We have conducted a large-scale study of deduplicated backup storage systems to discern their main characteristics. The study looks both broadly at autosupport data from over 10,000 deployed systems and in depth at content metadata snapshots from a few representative systems. The broad study examines filesystem characteristics such as file sizes, ages and churn rates while the detailed study focuses on deduplication and caching effectiveness. We contrast these results with those of primary filesystems from Microsoft [22].

As can be seen from §4, backup filesystems tend to have fewer, larger and shorter-lived files. Backups typically comprise either large repositories, such as databases, or large concatenations of protected files (e.g., tarfiles). As backup systems ingest these primary data stores on a repeating schedule they must delete and clean an equal amount of older data to maintain within capacity limits. This high data churn, averaging 21% of total storage per week leads to some unique demands of backup storage. They must sustain high write throughput and scale as primary capacity grows. This is not a trivial task as primary capacity scales with Kryder's law (about 100x per decade) but disk, network, and interconnect throughput have not scaled nearly as quickly [13]. To keep up with such workloads requires data reduction techniques, with deduplication being an important component of any data protection system. Additional techniques for reducing the ingest to a backup system, such as change-block tracking, are also important as systems scale further.

Backup workloads have two properties that help meet these challenging throughput demands. One is that the data is highly redundant between full backups. The other is that the data exhibits a lot of stream locality; that is, neighboring chunks of data tend to remain nearby across backups [34]. As seen in §5.2, leveraging these two qualities allows for very efficient caching, with deduplication hit rates of about 90% (including compulsory misses from new chunks).

Another interesting point is that backup storage workloads typically have higher demands for writing than reading. Primary storage workloads, which have less churn and longer-lived data, are skewed to relatively more read than write workload (2:1 as a typical metric [20]). However backup storage must be able to efficiently support read workloads, as well, to process efficient restores when needed and to replicate data offsite for disaster recovery. Optimizing for reads requires a more sequential disk layout and can be at odds with high deduplication rates, but effective backup systems must balance between both demands, which is an interesting area of future work.

The shift from tape-based backup to disk-based, purpose-built backup appliances has been swift and continues at a rate of almost 80% annually. By 2015 it is projected that disk-based deduplicating appliances will protect over 8EB of data [16]. Scaling write throughput at the same rate as data is growing, optimizing data layout, and providing efficiencies in capacity usage are challenging and exciting problems. The workload characterizations presented in this paper are a first step at better understanding a vital, unique, and under-served area in file systems research and we hope that it will stimulate further exploration.

Acknowledgments

We thank Stephen Manley, Hyong Shim, and Florin Sultan for helpful comments on earlier drafts. We are especially grateful to Bill Bolosky for sharing the statistical spreadsheet from the Microsoft FAST 2011 study, and to the anonymous referees and our shepherd, Florentina Popovici, for their feedback and guidance. We very much appreciate the many people within and outside EMC who enabled the metadata collection, particularly Hugo Patterson and Edgard Capdevielle.

References

[1] N. Agrawal, W. Bolosky, J. Douceur, and J. Lorch. A five-year study of file-system metadata. In *FAST'07: Proceedings of 5th Conference on File and Storage Technologies*, pages 31–45, February 2007.

[2] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, Oct. 1991.

[3] J. Bennett, M. Bauer, and D. Kinchlea. Characteristics of files in NFS environments. In *SIGSMALL'91: Proceedings of 1991 Symposium on Small Systems*, June 1991.

[4] D. R. Bobbarjung, S. Jagannathan, and C. Dubnicki. Improving duplicate elimination in storage systems. *Transactions on Storage*, 2:424–448, November 2006.

[5] W. J. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur. Single instance storage in Windows 2000. In *Proceedings of the 4th conference on USENIX Windows Systems Symposium - Volume 4*, pages 2–2, Berkeley, CA, USA, 2000. USENIX Association.

[6] M. Chamness. Capacity forecasting in a backup storage environment. In *LISA'11: Proceedings of the 25th Large Installation System Administration Conference*, Dec. 2011.

[7] A. Chervenak, V. Vellanki, and Z. Kurmas. Protecting file systems: A survey of backup techniques. In *Joint NASA and IEEE Mass Storage Conference*, 1998.

[8] W. Dong, F. Douglis, K. Li, H. Patterson, S. Reddy, and P. Shilane. Tradeoffs in scalable data routing for deduplication clusters. In *FAST'11: Proceedings of 9th Conference on File and Storage Technologies*, Feb. 2011.

[9] J. Douceur and W. Bolosky. A large scale study of file-system contents. In *SIGMETRICS'99: Proceedings of 1999 Conference on Measurement and Modeling of Computer Systems*, May 1999.

[10] C. Dubnicki, G. Leszek, H. Lukasz, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki. HYDRAstor: a scalable secondary storage. In *FAST'09: Proceedings of the 7th conference on File and Storage Technologies*, pages 197–210, February 2009.

[11] EMC Corporation. Data Domain Boost Software, 2010. `http://www.datadomain.com/products/dd-boost.html`.

[12] EMC Corporation. Data Domain products. `http://www.datadomain.com/products/`, 2011.

[13] J. Gantz and D. Reinsel. Extracting value from chaos. IDC Iview, available at `http://www.emc.com/collateral/analyst-reports/idc-extracting-value-from-chaos-ar.pdf`, June 2011.

[14] F. Guo and P. Efstathopoulos. Building a high-performance deduplication system. In *Proceedings of the 2011 USENIX conference on USENIX Annual Technical Conference*, 2011.

[15] W. Hsu and A. J. Smith. Characteristics of I/O traffic in personal computer and server workloads. *IBM Systems Journal*, 42:347–372, April 2003.

[16] IDC. Worldwide purpose-built backup appliance 2011-2015 forecast and 2010 vendor shares, 2011.

[17] E. Kruus, C. Ungureanu, and C. Dubnicki. Bimodal content defined chunking for backup streams. In *FAST'10: Proceedings of the 8th Conference on File and Storage Technologies*, February 2010.

[18] P. Kulkarni, F. Douglis, J. LaVoie, and J. M. Tracey. Redundancy elimination within large collections of files. In *Proceedings of the USENIX Annual Technical Conference*, pages 59–72, 2004.

[19] D. A. Lelewer and D. S. Hirschberg. Data compression. *ACM Computing Surveys*, 19:261–296, 1987.

[20] A. Leung, S. Pasupathy, G. Goodson, and E. L. Miller. Measurement and analysis of large-scale network file system workloads. In *Proceedings of the 2008 USENIX Technical Conference*, June 2008.

[21] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble. Sparse indexing: large scale, inline deduplication using sampling and locality. In *FAST'09: Proceedings of the 7th Conference on File and Storage Technologies*, pages 111–123, 2009.

[22] D. Meyer and W. Bolosky. A study of practical deduplication. In *FAST'11: Proceedings of 9th Conference on File and Storage Technologies*, February 2011.

[23] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom. SFS: Random write considered harmful in solid state drives. In *FAST'12: Proceedings of the 10th Conference on File and Storage Technologies*, 2012.

[24] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *SOSP'01: Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 174–187, 2001.

[25] Y. Nam, G. Lu, N. Park, W. Xiao, and D. H. C. Du. Chunk fragmentation level: An effective indicator for read performance degradation in deduplication

storage. In *Proceedings of the 2011 IEEE International Conference on High Performance Computing and Communications*, HPCC'11, pages 581–586, Washington, DC, USA, 2011. IEEE Computer Society.

[26] J. Ousterhout, H. DaCosta, D. Harrison, J. Kuntze, M. Kupfer, and J. G. Thompson. A trace driven analysis of the unix 4.2 BSD file system. *Proceedings of the Tenth Symposium on Operating Systems Principles*, Oct. 1985.

[27] N. Park and D. J. Lilja. Characterizing datasets for data deduplication in backup applications. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)*, 2010.

[28] C. Policroniades and I. Pratt. Alternatives for detecting redundancy in storage systems data. In *Proceedings of the USENIX Annual Technical Conference*, pages 73–86, 2004.

[29] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *FAST'02: Proceedings of the 1st USENIX conference on File and Storage Technologies*, 2002.

[30] D. Roselli, J. Lorch, and T. Anderson. A comparision of file system workloads. In *Proceedings of 2000 USENIX Annual Technical Conference*, June 2000.

[31] M. Satyanarayanan. A study of file sizes and functional life-times. In *SOSP'81: Proceedings of 8th ACM Symposium on Operating Systems Principles*, December 1981.

[32] P. Shilane, M. Huang, G. Wallace, and W. Hsu. WAN optimized replication of backup datasets using stream-informed delta compression. In *FAST'12: Proceedings of the 10th Conference on File and Storage Technologies*, 2012.

[33] Y. Tan, D. Feng, F. Huang, and Z. Yan. SORT: A similarity-ownership based routing scheme to improve data read performance for deduplication clusters. *IJACT*, 3(9):270–277, 2011.

[34] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *FAST'08: Proceedings of the 6th Conference on File and Storage Technologies*, pages 269–282, February 2008.

WAN Optimized Replication of Backup Datasets Using Stream-Informed Delta Compression

Philip Shilane, Mark Huang, Grant Wallace, and Windsor Hsu
Backup Recovery Systems Division
EMC Corporation

Abstract

Replicating data off-site is critical for disaster recovery reasons, but the current approach of transferring tapes is cumbersome and error-prone. Replicating across a wide area network (WAN) is a promising alternative, but fast network connections are expensive or impractical in many remote locations, so improved compression is needed to make WAN replication truly practical. We present a new technique for replicating backup datasets across a WAN that not only eliminates duplicate regions of files (deduplication) but also compresses *similar* regions of files with delta compression, which is available as a feature of EMC Data Domain systems.

Our main contribution is an architecture that adds stream-informed delta compression to already existing deduplication systems and eliminates the need for new, persistent indexes. Unlike techniques based on knowing a file's version or that use a memory cache, our approach achieves delta compression across all data replicated to a server at any time in the past. From a detailed analysis of datasets and hundreds of customers using our product, we achieve an additional 2X compression from delta compression beyond deduplication and local compression, which enables customers to replicate data that would otherwise fail to complete within their backup window.

1 Introduction

Creating regular backups is a common practice to protect against hardware failures and user error. To protect against site disasters though, replicating backups to a remote repository is necessary. Shipping tapes has been a common practice but has the disadvantages of being cumbersome, open to security breaches, and difficult to verify success. Replicating across the WAN is a promising alternative, but high-speed network connectivity is expensive and has been reserved mainly for Tier 1, primary data, which has not been available for backup replication.

Moreover, WAN bandwidth has not increased with data growth rates. While we tend to think of important data residing in corporate centers or data warehouses, computation has become pervasive and valuable data is increasingly generated in remote locations such as ships, oil platforms, mining sites, or small branch offices. Network connectivity may either be expensive or only available at low bandwidths.

Since network bandwidth across the WAN is often a limiting factor, compressing data before transfer improves effective throughput. More data can be protected within a backup window, or, for the same reasons, data is protected against disasters more quickly. Numerous systems have explored data reduction techniques during network transfer including deduplication [14, 25, 35, 37], which is effective at replacing identical data regions with references. A promising technique to achieve additional compression is delta compression, which compresses relative to similar regions by calculating the differences [17, 19, 36].

For both deduplication and delta compression, the goal is to find previous data that is either a duplicate or similar to data being transferred. We would like the pool of eligible data to include previous versions, maximizing our potential compression gains. A standard approach is to use a full index across the entire dataset, which requires space on disk, disk I/O, and ongoing updates [1, 19]. An alternative is to use a partial index holding data that has recently been transferred, which removes the persistent structures but shrinks the pool of eligible data [35]. Depending on the backup cycle, a week's worth of data or more may have to reside in an index to achieve much compression. We present a novel technique called Stream-Informed Delta Compression that achieves identity and delta compression across petabyte backup datasets with no prior knowledge of file versions while also reducing the index overheads of supporting both compression techniques.

Repeated patterns in backup datasets have been leveraged to design effective caching strategies to minimize disk accesses for deduplication [2, 16, 20, 23, 39, 41]. Their key observation is that for backup workloads, current data streams tend to have patterns that correspond to an earlier stream, which can be leveraged for effective caching. Our investigations show that the same data patterns exist for identifying similar data as well as duplicates, without additional index structures.

Our technique assumes that backup data is stored in a deduplicated format on both the backup server and remote backup repository. As streams of data are written to the backup server, they are divided into content-defined chunks, a secure fingerprint is calculated over each chunk, and only non-duplicate chunks are stored in containers devoted to that particular stream.

We augment this standard technique by calculating a *sketch* of each non-duplicate chunk. Sketches, sometimes referred to as resemblance hashes, are weak hashes of the chunk data with the property that if two chunks have the same sketch they are likely near-duplicates. These can be used during replication to identify similar (non-identical) chunks. Instead of using a full index mapping sketches to chunks, we rely on the deduplication system to load a cache with sketches from a previous stream, which we demonstrate in Section 6 leads to compression close to using a full sketch index. During replication, chunks are deduplicated, and non-duplicate chunks are delta compressed relative to similar chunks that already reside at the remote repository. We then apply GZ [15] compression to the remaining bytes and transfer across the WAN to the repository where delta compressed data is first decoded and then stored.

There are several important properties of Stream-Informed Delta Compression. First, we are able to achieve delta compression against any data previously stored and are not limited to a single identified file or the size constraints of a partial index. Since delta compression relies upon a deduplication system to load a cache, there is a danger of missing potential compression, but our experiments demonstrate the loss is small and is a reasonable trade-off.

Second, our architecture only requires one index of fingerprints, while traditional similarity detection required one or more on-disk indexes for sketches [1, 19] or used a partial index with a decrease in compression. Another important consideration in minimizing the number of indexes is that updating the index during file deletion is a complicated step, and reducing complexity/error cases is important for production systems.

Our delta compression algorithm has been released commercially as a standard feature for WAN replication between Data Domain systems. Customers have the option of turning on delta compression when replicating between their deduplicated backup storage systems to achieve higher compression and correspondingly higher effective throughput. Analyzing statistics from hundreds of customers in the field shows that delta compression adds an additional 2X compression and enables the replication of more data across the WAN than could otherwise be protected.

2 Similarity Index Options

To achieve the highest possible compression during WAN replication, we would like to find similarity matches across the largest possible pool of chunks. While previous projects have delta encoded data for replication, the issue of indexing sketches efficiently has not been explored. In this section, we discuss tradeoffs for three indexing options.

2.1 Full Sketch Index

The conceptually simplest solution is to use a full index mapping from sketch to chunk. Unfortunately, for terabytes or petabytes of storage, the index is too large for memory and must be kept on disk, though several previous projects have used a full index for storing sketches [1, 18, 19, 40]. As an example, for a production deduplicated storage system with 256 TB of capacity, 8 KB average chunk size, and 16 bytes per record, the sketch index would be a half-TB. Sketches are random values so there is little locality in an index system, and every query will cause a disk access.

Also, a common technique is for sketches to actually consist of subunits called *super-features* that are indexed independently [4, 19]. Using multiple super-features increases the probability of finding a similar chunk (see Section 4.1), but it also requires a disk access for each super-feature's on-disk index, followed by a disk access for the base chunk itself. Unless the number of disk spindles increases, lookups will be slowed by disk accesses. Another detail that is often neglected is that each index has to be updated as chunks are written and deleted from the system, which can be complicated in a live system. Moving the index to flash memory decreases lookup time [10] but increases hardware cost.

2.2 Partial Sketch Index

An alternative to a full index is to use a partial index holding recently transmitted sketches, which would probably reside in memory, but could also exist on disk. The advantage of a partial index is that it can be created as data is replicated without the need for persistent data structures, and several projects [33, 35] and products [32] use a cache structure. Sizing and updating a partial index are important considerations. The most common implementations are FIFO or LRU policies [33], which have the advantage of finding similar chunks nearby in the replication stream, but will miss

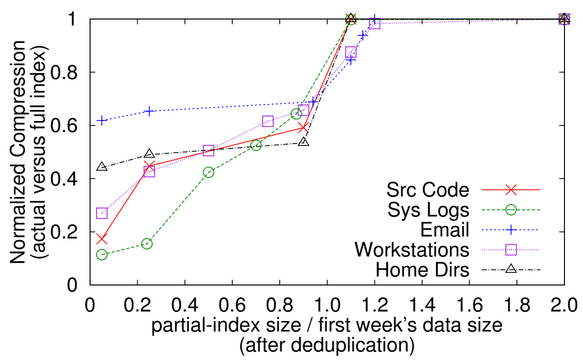


Figure 1: Optimal compression in a backup configuration (e.g. weekly full backup) requires an index to include at least a full backup cycle (1.0 on the x-axis).

distant matches. For backup workloads, repeated data may not appear until next week's full backup takes place, and enterprise organizations typically have hundreds to thousands of primary storage machines to be backed up within that time. Therefore, a partial index would have to be large enough to hold all of an organization's primary data. Riverbed [32] uses an array of disks to index recently transferred data.

Another form of a partial-index is to use version information. As an example, rsync [37] uses file pathnames as the mechanism to find previous versions to perform compression before network transfer.

We analyze this experimentally in Figure 1, which shows how much compression is achieved as index coverage increases (more details are in Section 6). The datasets consist of two weeks worth of backup data, and the combination of deduplication and delta compression across both weeks is presented, normalized relative to compression achievable with a full index (right-most data points). This result shows a sharp increase in compression aligned with the one week boundary when sufficient data are covered by an index for both deduplication and delta compression. Effectively, a partial index would have to be nearly as large as a full index to achieve high compression.

2.3 Stream-Informed Sketch Cache

Numerous papers have explored properties of backup datasets and found that there are repeated patterns related to backup policies. These patterns have been leveraged in deduplication systems to prefetch fingerprints written sequentially by a previous data stream [2, 16, 20, 39, 41]. We discovered that similarity detection has the same stream properties as deduplication, because small edits to a file will probably be a similarity match to the previous backup of the same file, and edits may be surrounded by duplicate regions that can load a cache effectively. This exploration of similarity locality is one of the major contributions of our work.

Following on previous work, we could build a cache and indexing system similar to deduplicating systems (i.e. Bloom filters and indexes), but a disadvantage of this approach is that the number of indexing structures increases with the number of super-features and adds complexity to our system.

Instead, we leverage the same cache-loading technique used by our storage system for deduplication [41]. While loading a previous stream's fingerprints into a cache, we also load sketches from the same stream. This has the significant advantage of removing the need for extra on-disk indexes that must be queried and maintained, but it also has the potential disadvantage of less similarity detection than indexing sketches directly.

To explore these alternatives, we built a full sketch index, a partial index, and a stream-informed cache that piggy-backs on deduplication infrastructure. In Section 6 we explore trade-offs between these three techniques.

3 Delta Replication Architecture

While our research has focused on improving the compression and throughput of replication, it builds upon deduplication features of Data Domain backup storage systems. We first present an overview of our efficient caching technique before augmenting that architecture to support delta compression in replication.

3.1 Stream-Informed Cache for Deduplication

A typical deduplication storage system receives a stream consisting of numerous smaller files concatenated together in a tar-like structure. The file is divided into content-defined chunks [22, 25], and a secure hash value such as SHA-1 is calculated over each chunk to represent it as a fingerprint. The fingerprint is then compared against an index of fingerprints for previously stored chunks. If the fingerprint is new, then the chunk is stored and the index updated, but if the fingerprint already exists, only a reference to the previous chunk is maintained in a file's meta data. Depending on backup patterns and retention period, customers may experience 10X or higher deduplication (logical file size divided by post-deduplication size).

Early deduplication storage systems ran into a fingerprint index bottleneck, because the index was too large to fit in memory, and index lookups limited overall throughput [30]. Several systems addressed this problem by introducing caching techniques. The key insight of the Data Domain system [41] is that when a fingerprint is a duplicate, the following fingerprints will likely match data written consecutively in an earlier stream. We present our basic deduplication architecture along with highlighted modifications in Figure 2. Fingerprints

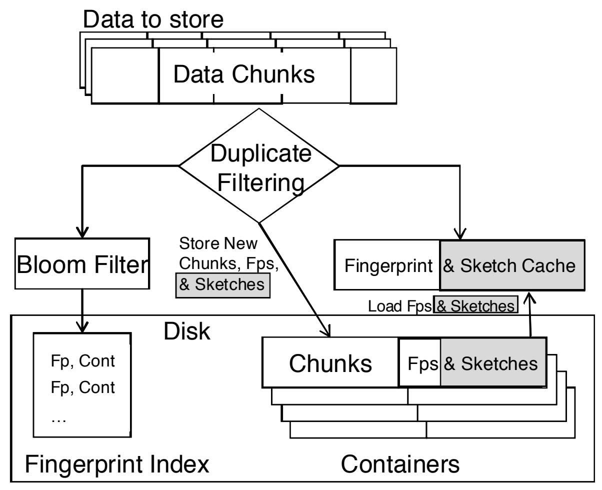


Figure 2: Data Domain deduplication architecture with cache, Bloom filter, fingerprint index, and containers. Highlighted modifications show sketches stored in containers and loaded in a stream-informed cache when fingerprints are loaded.

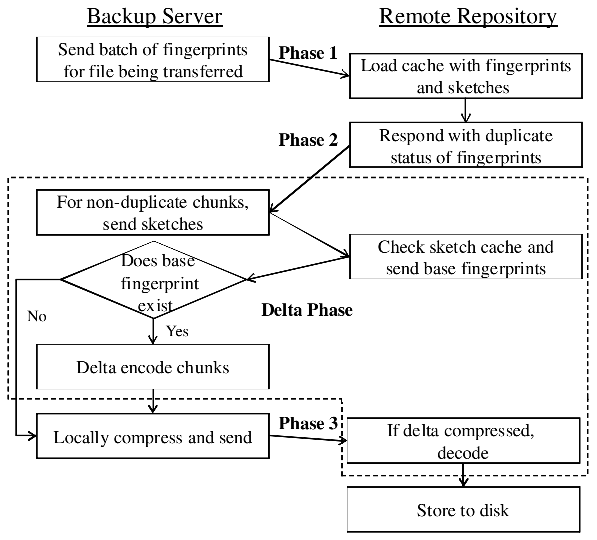

Backup Server
Remote Repository
Send batch of fingerprints for file being transferred
Phase 1
Load cache with fingerprints and sketches
Phase 2
Respond with duplicate status of fingerprints
For non-duplicate chunks, send sketches
Check sketch cache and send base fingerprints
Does base fingerprint exist
No
Yes
Delta Phase
Delta encode chunks
Locally compress and send
Phase 3
If delta compressed, decode
Store to disk

Figure 3: Replication protocol modified to include delta compression.

and chunks are laid out in containers and can be loaded into a fingerprint cache. When a chunk is presented for storage, its fingerprint is compared against the cache, and on a miss, a Bloom filter is checked to determine whether the fingerprint is likely to exist in an on-disk index. If so, the index is checked, and the corresponding container's list of fingerprints is loaded into the cache. When eviction occurs, based on an LRU policy, all fingerprints from a container are evicted as a group. Other techniques for maintaining fingerprint locality have been presented [2, 16, 20, 23, 39], which indexed either deduplicated chunks or the logical stream of file data.

3.2 Replication with Deduplication

For disaster recovery purposes, it is important to replicate backups from a backup server to a remote repository. Replication is a common feature in storage systems [28], and techniques exist to synchronize versions of a repository while minimizing network transfer [18, 37]. In most cases, these approaches result in completely reconstructing files at the destination.

For deduplication storage systems, it is natural to only transfer the unique chunks and the meta data needed to reconstruct logical files. Although not described in detail, products such as Data Domain BOOST [13] already support deduplicated replication by querying the remote repository with fingerprints and only transferring unique chunks, which can be compressed with GZ or other local compressors. Earlier work by Eshghi et al. [14] presented a similar approach that minimized network transfer by querying the remote repository with a hierarchical file consisting of hashes of chunks. These approaches removes duplicates in network-constrained environments.

3.3 Delta Replication

We expand upon standard replication for deduplication systems by introducing delta compression to achieve higher total compression than deduplication and local compression can achieve. We modified the basic architecture in Figure 2, adding sketches to the container meta data section. Sketches are designed so that similar chunks often have identical sketches. As data is written to a deduplicating storage node, non-duplicate chunks are further processed to create a sketch, which is stored in the container along with the fingerprint. During duplicate filtering at the repository, both fingerprints and sketches are loaded into a cache. In later sections, we explore trade-offs of this architecture decision.

3.4 Network Protocol Considerations for Delta Compression

The main issue to address is that both source and destination must agree on and have the same base chunk, the source using it to encode and the destination to decode. Figure 3 shows the protocol we chose for combining deduplication and delta compression. The backup server sends a batch of fingerprints to the remote repository, which loads its cache, performs filtering, and responds indicating which corresponding chunks are already stored. For delta compression, the backup server then sends the sketches of unique chunks to the repository, and the repository checks the cache for matching sketches. The repository responds with the fingerprint corresponding to the similar chunk, called the base fingerprint, or indicates that there is no similarity match. If the backup server has the base fingerprint, it delta compresses a chunk relative to the base before local compression and transfer. At the repository, delta encoded and compressed chunks are uncompressed and decoded in preparation for storage.

We considered sending sketches with fingerprints in Phase 1, but sending sketches after filtering (Phase 2) reduces wasted meta data overhead, compared to sending the sketches for all chunks. Fingerprint filtering occurs on the destination, and its cache is properly set up to find similar chunks. So in practice, it is best if the destination performs similarity lookup.

4 Implementation Details

In this section, we discuss: creating sketches, selecting a similar base chunk, and delta compression relative to a base.

4.1 Similarity Detection with Sketches

In order to delta compress chunks, we must first find a similar chunk already replicated. Numerous previous projects have used sketches to find similar matches, and our technique is most similar to the work of Broder et al. [4, 5, 6].

Intuitively, similarity sketches work by identifying "features" of a chunk that would not likely change even as small variations are introduced in the data. One approach is to use a rolling hash function over all overlapping small regions of data (e.g. 32 byte windows) and choose as the feature the maximal hash value seen. This can be done with multiple different hash functions generating multiple features. Chunks that have one or more features (maximal values) in common are likely to be very similar, but small changes to the data are unlikely to perturb the maximal values [4].

Figure 4 shows an example with data chunks 1 and 2 that are similar to each other and have four sketch features (maximal values) in common. They have the same maximal values because the 32-byte windows that generated the maximal values were not modified by the added regions (in red). If different regions had changed it could affect one or more of the maximal values, so different maximal features would be selected to represent chunk 2. This would cause a feature match to fail. In general, as long as some set of the maximal values are unchanged, a similarity match will be possible.

For our sketches we group multiple features together to form "super-features" (also called super-fingerprints in [19]). The super-feature value is a strong hash of the underlying feature values. If two chunks have an identical super-feature then all the underlying features match. Using super-features helps reduce false positives and requires chunks to be more similar for a match to be found.

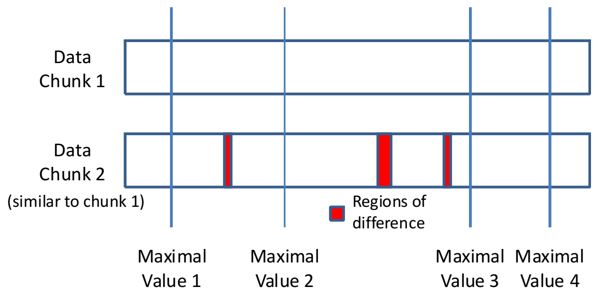


Figure 4: Similar chunks tend to have the same maximal values, which can be used to create features for a sketch.

To generate multiple, independent features, we first generate a Rabin fingerprint *Rabin_fp* over rolling windows w of chunk C and compare the fingerprint against a mask for sampling purposes. We then permute the Rabin fingerprint to generate multiple values with function π_i with randomly generated coprime multiplier and adder values m and a.

$$fp = Rabin\_fp(w)$$

$$\pi_i(fp) = (m_i * fp + a_i) \ mod \ 2^{32}$$

If the result of $\pi_i(fp)$ is maximal for all w, then we retain the Rabin fingerprint as $feature_i$. After calculating all features, a super-feature sf_j is formed by taking a Rabin fingerprint over k consecutive features. We represent consecutive features as $feature_{b...e}$ for beginning and ending positions b and e, respectively.

$$sf_j = Rabin\_fp(feature_{j*k...j*k+k-1})$$

As an example, to produce three super-features with $k = 4$ features each, we generate twelve features, and calculate super-features over the features 0...3, 4...7, and 8...11.

We performed a large number of experiments varying the number of features per super-feature and number of super-features per sketch. Increasing the number of features per super-feature increases the quality of matches, but also decreases the number of matches found. Increasing the number of super-features increases the number of matches but with increased indexing requirements. We typically found good similarity matches with four features per super-feature and a small number of super-features per sketch. These early experiments were completed with datasets that consisted of multiple weeks of backups and had sizes varying from hundreds of gigabytes to several terabytes. We explore the delta compression benefits of using more than one super-feature in Section 6.4.

To perform a similarity lookup, we use each super-feature as a query to an index representing the corresponding super-features of previously processed chunks.

Chunks that match on more super-features are considered better matches than those that match on fewer super-features, and experiments show a correlation between number of super-feature matches and delta compression. Other properties can be used when selecting among candidates including age, status in a cache, locality on disk, or other criteria.

4.2 Delta Compression

Once a candidate chunk has been selected, it is referred to as the *base* used for delta compression, and the *target* chunk currently being processed will be represented as a 1-level delta of the base. To perform delta encoding, we use a technique based upon Xdelta [21] which is optimized for compressing highly similar data regions.

We initialize the encoding by iterating through the base chunk, calculating a hash value at subsampled positions, and storing the hash and offset in a temporary index. We then begin processing the target chunk by calculating a hash value at rolling window positions. We look up the hash value in the index to find a match against the base chunk. If there is a match, we compare bytes in the base and target chunks forward and backward from the starting position to create the longest match possible, which is encoded as a `copy` instruction. If the bytes fail to match, we issue an `insert` instruction to insert the target's bytes into the output buffer, and we also add this region to the hash index. During the backward scans, we may intersect a region previously encoded. We handle this by determining whether keeping the previous instruction or updating it will lead to greater compression. Since we are performing delta compression at the chunk level, as compared to the file level, we are able to maintain this temporary index and output buffer in memory.

5 Experimental Details

We perform actual replication experiments on working hardware with multi-month datasets whenever practical, but we also use simulators to compare alternative techniques. In this section, we first present the datasets tested, then details of our experimental setup, and finally compression metrics.

5.1 Datasets

In this paper we use backup datasets collected over several months as shown in Table 1, which lists the type of data, total size in TB, months collected, deduplication, delta, GZ, and total compression. Total compression is measured as data bytes divided by replicated bytes (after all types of compression) and is equivalent to the multiplication of deduplication, delta, and GZ. For the compression values, we used results from our default configuration. These datasets were previously studied for deduplication [11, 27] but not delta compression. Note that our deduplication results vary slightly (within 5%) from Dong et al. [11] due to implementation differences.

We also highlight steady-state delta compression after a seeding period has completed. For all of the datasets except `Email`, seeding was one week, and the period after seeding is the remaining months of data. Customers often handle initial seeding by keeping pairs of replicating machines on a LAN (when new hardware is installed) until seeding completes and then move the destination machine to the long-term location. Alternatively, seeding can be handled using backups available at the destination. While there is some delta compression within the seeding period, delta compression increases once a set of base chunks become available, and the period after seeding is indicative of what customers will experience for the lifetime of their storage.

These datasets consist of large "tar" type files representing many user files or objects concatenated together by backup software. Except for `Email` (explained below), these datasets consist of a repeated pattern of a weekly full backup followed by six, smaller incremental backups.

Source Code Repository: Backups from a version control repository containing source code.

Workstations: Backups from 16 desktops used by software engineers.

Email: Backups from a Microsoft Exchange server. Unlike the other datasets, `Email` consists of daily full backups, and the seeding phase consists of a single backup instead of a week's worth of data.

System Logs: Backups from a server's /var directory, mostly consisting of emails stored by a list server.

Home Directories: Backups from software engineers' home directories containing source code, office documents, etc.

5.2 Delta Replication Experiments

Many of our experiments were performed on production hardware replicating between pairs of systems in our lab. We actually used a variety of machines that varied in storage capacity (350 GB - 5 TB), RAM (4 GB - 16 GB), and computational resources (2 - 8 cores). We have controlled internal parameters and confirmed that disparate machines produce consistent results. Unless specifically stated, we ran all experiments with 3 super-features per sketch, 12 MB sketch cache, 8 KB average chunk size, and 4.5 MB containers holding meta data and locally compressed chunks. When applying local compression, we create compression regions of approximately 128 KB of chunks.

5.3 Simulator Experiments

We compare our technique of replication with a fingerprint index and sketch cache against two alternative ar-

Name	Properties		Entire Dataset				Seeding	After Seeding			
	TB	Months	Dedupe	Delta	GZ	Total	GB	Dedupe	Delta	GZ	Total
Source Code	4.6	6	20.25	2.97	3.24	194.86	140	24.91	3.75	3.99	372.72
Workstations	4.9	6	5.62	4.44	1.93	48.16	166	5.70	4.62	1.91	50.30
Email	5.2	7	6.79	1.95	2.95	39.06	16	6.90	1.97	2.96	40.24
System Logs	5.4	4	37.19	2.39	2.38	211.54	254	57.94	3.55	2.86	588.26
Home Dirs	12.9	3	19.20	1.90	1.48	53.99	491	31.66	2.89	1.91	174.76

Table 1: Summary of datasets. Deduplication, delta, and GZ compression factors are shown across the entire dataset as well as for the period after seeding, which was typically one week.

chitectures: 1) full fingerprint and sketch indexes and 2) a partial-index of fingerprints and sketches implementing an LRU eviction policy.

Before building the production system, we actually started with a simplified simulator that maintained a full index of fingerprints and sketches in memory. To decrease memory overheads, we use 12 bytes per fingerprint as compared to larger fingerprints necessary for a product such as a 20 byte SHA-1. In a separate analysis, we found that 12 byte fingerprints only cause a small number of collisions out of the hundreds of millions of chunks processed. To maximize throughput and simplify the code, we try to keep the entire index in RAM. Also, instead of implementing a full replication protocol, we record statistics as the client deduplicates and delta compresses chunks without network transfer. Our simulator did not apply local compression with the same technique as our replication system, so comparisons to the simulator do not include local compression.

Our second simulator explores the issues of data locality and index requirements with an LRU partial-index of fingerprints and sketches. This partial-index is a modification of the previous simulator with the addition of parameters to control the index size. The partial-index only holds meta data, fingerprints and sketches, which each reference chunks stored on disk. The fingerprint and sketches for a chunk maintain the same age in the partial-index, so they are added and evicted as a unit. If a fingerprint is referenced as a duplicate of incoming data or a sketch is selected as the best similarity match for compression, the age is updated.

5.4 Compression Metrics

Our focus is on improving replication across the WAN, specifically for customers with low network connectivity. For that reason, we mostly focus on compression metrics, though we also present throughput results from experiments and hundreds of customer systems.

We tend to use the term *compression* generically to refer to any type of data reduction during replication such as deduplication, delta compression, or local compression with an algorithm such as GZ. Compression is calculated as *original_bytes/post_compression_bytes*. However, we generally use the term *total compression* to mean data reduction achieved by deduplication, delta, and GZ in combination. As an example, if the deduplication factor is $10X$, delta is $2X$, and GZ is $1.5X$ then total compression is $30X$ since these techniques have a multiplicative effect. A compression factor of $1X$ indicates no data reduction. In order to show different datasets on the same graph, we often plot *normalized compression*, which is total compression of a particular experiment divided by the maximum total compression. As explained in Section 6, maximum compression is measured using a full index or the appropriate baseline for each experiment and dataset. Normalized compression is in the range $(0...1]$.

6 Results

In this section, we begin by exploring parameters of our system (cache size, number of super-features, and multi-level delta) and then compare Stream-Informed Delta Compression to alternative techniques such as using a full sketch index or maintaining a partial-index of recently used sketches. We then investigate the interaction of delta and GZ compression.

6.1 Sketch Cache Size

When designing our cache-based delta system, sizing the cache is an important consideration. If datasets have similarity locality that matches up perfectly to deduplication locality, then a cache holding a single container could theoretically achieve all of the possible compression. With a larger cache, similarity matches may be found to chunks loaded in the recent past, with compression growing with cache size. We found that the hit rate is maximized with a cache sized consistently across datasets even though Home Directories is over twice as large as the other datasets.

We evaluated the sketch cache hit rate in Figure 5, by increasing the sketch cache size (x-axis) and measuring the number of similarity matches found in the cache relative to using a full index. The sketch cache size refers to the amount of memory required to hold sketches, which is approximately 12 bytes per super-feature. Therefore a cache of 12 MB corresponds to 1 million super-features

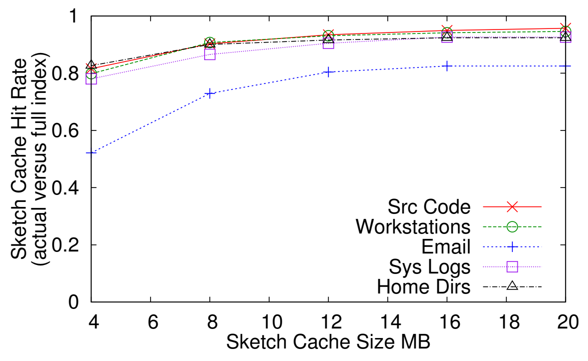


Figure 5: Locality-informed sketch cache hit rate reaches its maximum with a cache of 12-16 MB.

and 1/3 million chunks, since we have 3 super-features per sketch by default.

With a cache of 4 MB, the hit rate is between 50% and 90% of the maximum, and the hit rate grows until around 12 or 16 MB, when it is quite close to the final value we show at 20 MB. `Email` showed the worst hit rate, maxing at around 80%, which is still a reasonably high result. `Email` has worse deduplication locality than the other datasets and this impacts delta compression in a data-dependent manner. Regardless of the dataset size (5 TB up to 13 TB) and deduplication (5-37X), all of the datasets reached their maximum hit rates with a similarly sized cache. Our implementation has a minimum cache size related to the large batches of chunks transferred during replication as well as the multiple stages of pipelined replication that either add data to the cache or need to check for matches in the cache.

Although it may be reasonable to use a larger cache in enterprise-sized servers, note that our experiments are for single datasets at a time. A storage server would normally handle numerous simultaneous streams, each needing a portion of the cache, so our single-stream results should be scaled accordingly. Since the locality of delta compression for backup datasets corresponds closely to identity locality, only a small cache is needed, and our memory requirements should scale well with the number of backup streams. Our intuition is that users/applications often make small modifications to files, so duplicate chunks indicate a region of the previous version of a file that is likely to provide delta compression.

6.2 Delta Encoding

Our similarity detection technique is able to find matches for most chunks during replication and achieves high encoding compression on those chunks. The second column of Table 2 shows the percentage of bytes after deduplication that are delta encoded after seeding. 55-82% of bytes undergo delta encoding with a median of 77%. Delta encoding factors vary from 8.91-30.11X with a median of 15.65X. As an example of how the delta factor is calculated for `System Logs`, 77% of bytes after deduplication are delta encoded to $\frac{1}{15.65}$ of their original size, and 23% of bytes are not encoded. Therefore, $\frac{1}{\frac{.77}{15.65}+.23} \approx 3.55$ (rounding in the tables affects accuracy), which is equivalent to dividing post-deduplication bytes by post-delta compression bytes.

Name	% Post-Dedupe Bytes	Encoding Factor	Delta Factor
Source Code	82	8.91	3.75
Workstations	81	30.05	4.62
Email	55	10.05	1.97
System Logs	77	15.65	3.55
Home Dirs	68	30.11	2.89
Median	77	15.65	3.55

Table 2: Datasets, percent of post-deduplication bytes delta encoded, delta encoding factor, and resulting delta factor for each dataset, which corresponds to Table 1 after seeding.

While further improvements in encoding compression are likely possible, we are already shrinking delta encoded chunks to a small fraction of their original size. On the other hand, increasing the fraction of chunks that receive delta encoding could lead to larger savings.

6.3 Multi- vs 1-Level Delta

While we have described the delta compression algorithm as representing a chunk as a 1-level delta from a base, because we decode chunks at the remote repository, our delta replication is actually multi-level. Specifically, consider a delta encoded chunk B transferred across the network that is then decoded using base chunk C and stored. At a later time, another delta encoded chunk A is transferred across the network that uses B as a base. Although B exists in a decoded form, it was previously a 1-level delta encoded chunk, so A is effectively a 2-level delta because A referenced B, which referenced C. Our replication system, like many, does not bound the delta level, since chunks are decoded at the destination, and we effectively achieve multi-level delta across the network.

As compared to replicating delta compressed chunks, storing such chunks introduces extra complexity. Although n-level delta is possible for any value of n, decoding an n-level delta entails n reads of the appropriate base chunks, which can be inefficient in a storage system. For this reason, a delta storage system [1] may only support $1-$ or 2-level delta encodings to bound decode times.

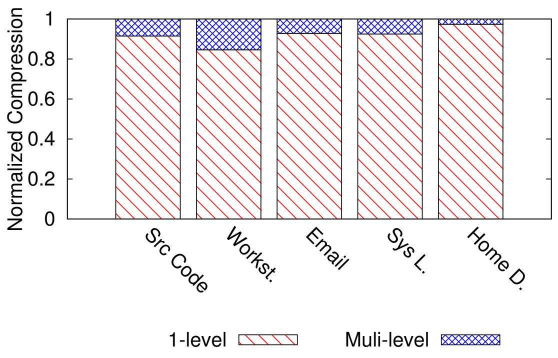


Figure 6: Multi-level delta compression improves 6-30% beyond 1-level delta.

To compare the benefits of multi- and 1−level delta, we studied the compression differences. We modified our replication system so that after a chunk is delta encoded, its sketch is then invalidated. This ensures that delta encoded chunks will never be selected as the base for encoding other chunks, preventing 2-level or higher deltas.

In Figure 6, multi- and 1−level delta are compared, with multi-level delta adding 1.03 - 1.18X additional compression. As an example, `Source Code` increased from 178X to 194X total compression (deduplication, delta, and GZ), which is roughly similar to adding a second super-feature as discussed in Section 6.4. These results also highlight that 1-level delta is a reasonable approximation to multi-level, when multi-level is impractical. Unlike a storage system, we are able to get the compression benefits of multi-level without the slowdowns related to decoding *n*-level delta chunks.

6.4 Sketch Index vs Stream-Informed Sketch Cache

We next investigate how our stream-informed caching technique compares to the alternative of a full sketch index. We expect that using a full sketch index could find potential matches that a sketch cache will miss because of imperfect locality, but maintaining indexes for billions of stored chunks adds significant complexity. We explore the compression trade-offs by comparing delta replication with a cache against a simulator with complete indexes for each super-feature.

Figure 7 compares compression results for the index and cache options. The lowest region of each vertical bar is the amount of compression achieved by deduplication, and because of differences in implementation between our product and simulator, these numbers vary slightly. The next four sets of colored regions show how much extra compression is achieved by using 1-4 super-features. The cache experiments ran on production hardware, and the cache was fixed at 12 MB. Also, our simulator with

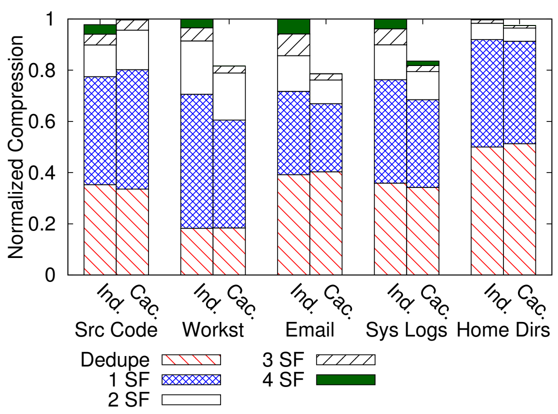


Figure 7: Using a stream-informed sketch cache results in nearly as much compression as using a full index, and using two super-features with a cache achieves more compression than a single super-feature index.

index did not apply local compression, so only deduplication and delta compression are analyzed.

In all cases, using a single super-feature adds significant compression beyond deduplication alone, with decreasing benefit as the number of super-features increases. Although using a sketch cache generally has lower delta compression than an index, the results are reasonably close (`Workstations` with 1 super-feature and a cache is within 14% of the index with 1 super-feature). Importantly, we can use more than one super-feature in our cache with little additional overhead compared to multiple on-disk indexes for super-features. Using a cache with two or more super-features achieves greater compression than a single index, which is why we decided to pursue the caching technique.

An interesting anomaly is that `Source Code` achieved higher delta compression with a stream-informed sketch cache than a full index, even though we would expect a limited-size cache to be an approximation to a full index. We found that `Source Code` and `Home Directories` had extremely high numbers of potential similarity matches ($> 10,000$) all with the same number of super-feature matches, which was likely due to repeated headers in source files[1]. Selecting among the candidates leads to differences in delta compression, and the selection made by a stream-informed cache leads to higher compression for `Source Code` than our tie-breaking technique for the index (most recently written).

[1]This caused slowed throughput for `Home Directories`, and those experiments would not have completed without adjusting the sketch index. We modified the sketch index for all `Home Directories` results such that if a sketch has more than 128 similarity matches, the current sketch is not added to the index.

`Home Directories` had similar compression with either a cache or index.

Another unexpected result is that increasing the number of super-features used with our cache did not always increase total compression. Since we fix the size of our cache at 12 MB, when the number of super-features increases, fewer chunks are represented in the cache. The optimal cache size tends to increase with the number of super-features, but the index results indicate that adding super-features has diminishing benefit.

6.5 Partial-index of Fingerprints and Sketches

As a comparison to previous work, we implemented a partial-index of fingerprints and sketches that updates ages when either a chunk's fingerprint or sketch is referenced and evicts from the partial-index with an LRU policy. While it is somewhat unfair to compare a partial-index to our technique, it is useful for analyzing the scalability of such systems.

To focus on the data patterns of typical backups, we limit this experiment to two full weeks of each dataset, which typically consists of a full backup followed by six incremental backups followed by another full and six incremental backups. For `Email`, we selected two full backups a week apart, since a full backup was created each day.

Figure 1 (presented in Section 2) shows the amount of compression achieved (deduplication and delta) as the partial-index size increases along the x-axis, which is measured as the fraction of the first week's data kept in a partial-index. When the partial-index is able to hold more than a week's worth of data (1.0 on x-axis), compression jumps dramatically as the second week's data compresses against the first week's data. To highlight this property, the horizontal axis is normalized based on the first week's deduplication rate, since the post-deduplication size affects how many fingerprints and sketches must be maintained.

These results highlight that techniques using a partial-index must hold a full backup cycle's worth of data (e.g. at least one full backup) to achieve significant compression, while our delta compression technique uses a combination of a deduplication index and stream-informed sketch cache to achieve high compression with small memory overheads. For storage systems with large backups or backups from numerous sources, our algorithm would tend to scale memory requirements better, since Figure 5 demonstrates that we only need a fixed-size cache regardless of the dataset size.

6.6 Interaction of Delta and Local Compression

Our replication system includes local compressors such as GZ that can be selected by the administrator. During replication, chunks are first deduplicated and many of the remaining chunks are delta compressed. All remaining data bytes (delta compressed or not) are then compressed with a local compressor. A subtle detail of delta compression is that it reduces redundancies within a chunk that appear in the previous base chunk and within itself, which overlaps with compression that local compressors might find.

Name	No Delta	With Delta		Delta
	GZ	Delta	GZ	Improve.
Source Code	7.20	3.75	3.99	2.08
Workstations	2.83	4.62	1.91	3.12
Email	3.12	1.97	2.96	1.87
System Logs	4.63	3.55	2.86	2.19
Home Dirs	3.12	2.89	1.91	1.77
Median	3.12	3.55	2.86	2.08

Table 3: Delta encoding overlaps with the effectiveness of GZ, but total compression including delta is still a 2X improvement beyond alternative approaches. Results are after initial seeding.

We evaluated the impact of delta compression on GZ and total compression by rerunning our replication experiments with GZ enabled and delta compression either enabled or disabled. Table 3 shows GZ compression achieved both with and without delta after seeding. Results with delta enabled are the same as Table 1. Deduplication factors are the same with or without delta enabled, and are removed from the table for space reasons. GZ and delta overlap by 5-50% (7.20X vs 3.99X for GZ on `Source Code`), but using delta in combination with GZ still provides improved total compression (2.08X for `Source Code`). The overlap of local compression and delta compression varies with dataset and type of local compressor selected (GZ, LZ, etc.), but we typically see significant advantages to using both techniques in combination with deduplication.

6.7 WAN Replication Improvement

We performed numerous replication experiments measuring network and effective throughput. Figure 8 shows a representative replication result for the `Workstations` dataset. Throughput was throttled at T3 speed (44 Mb/s) and measured every 10 minutes. We found effective throughput is 1-2 orders of magnitude faster than network throughput, which corresponds to total compression. Although throughput could be further improved with better pipelining and buffering, this result highlights that compression boosts effective throughput and reduces the time until transfer is complete.

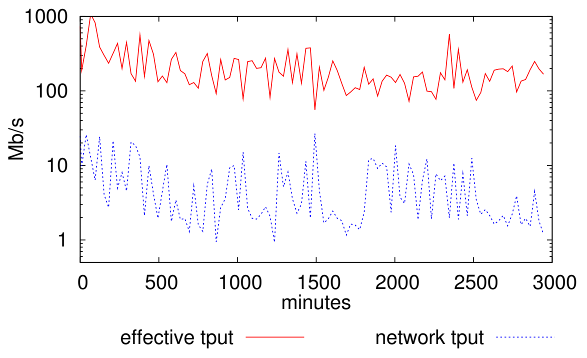


Figure 8: Effective throughput is higher than network throughput due to compression during replication.

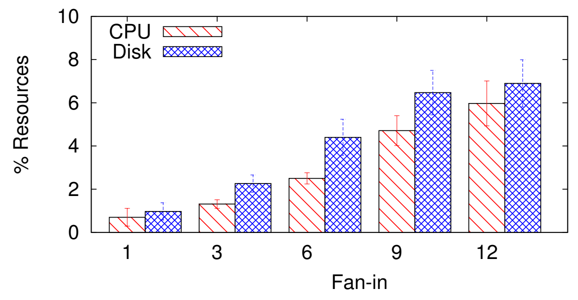


Figure 9: CPU and disk utilization grows fairly linearly on the remote repository as the number of replication streams increases. Error bars indicate a standard deviation.

7 Performance Characteristics

In this section, we discuss overheads of delta compression and limitations of stream-informed delta compression.

7.1 Delta Overheads

First, capacity overheads for storing sketches are relatively small. Each chunk stored in a container (after deduplication) also has a sketch added to the meta data section of the container, which is less than 20 bytes, but our stream-informed approach removes the need for a full on-disk index of sketches.

There are also two performance overheads added to the system: sketching on the write path and reading similar base chunks to perform delta compression. First, incoming data is sketched before being written to disk, which introduces a 20% slowdown in unoptimized tests. The sketching stage happens after deduplication, so after the first full backup, later backups experience less slowdown since a large fraction of the data is duplicate and does not need to be sketched. As CPU cores increase and pipelining is further optimized, this overhead may become negligible.

The second, and more sizable throughput overhead, is during replication when similar chunks are read from disk to serve as the base for delta compression, which limits our throughput by the read speed of our storage system. Our read performance varies with the number of disk spindles and data locality, which we are continuing to investigate. Remote sites also tend to have lower-end hardware with fewer disk spindles than data warehouses. For these reasons, we recommend turning on delta compression for low bandwidth connections (6.3 Mb/s or slower), where delta compression is not the bottleneck and extra delta compression multiplies the effective throughput. Also, it should be noted that read overheads only take place when delta compression occurs, so if no similarity matches are found, read overhead will be minimal.

Effectively, we are trading computation and I/O resources for higher network throughput, and we expect computation and I/O to improve at a faster rate than network speeds increase, especially in remote areas. We expect this tradeoff to become more important in the future as data sizes continue to grow. Improvements to our technique and hardware may also expand the applicability of delta replication to a larger range of customers.

Delta compression increases computational and I/O demands on both the backup server and remote repository. We set up an experiment replicating from twelve small backup servers (2 cores and 3-disk RAID) to a medium-sized remote repository (8 cores and 14-disk RAID) with a T1 connection (1.5 Mb/s). At the backup servers, the CPU and disk I/O overheads were modest (2% and 4% respectively). At the remote repository, CPU and disk overhead scaled linearly as the number of replication streams grew from 1 to 12 as shown in Figure 9. Measurements were made over every 30 second period after the seeding phase, and standard deviation error bars are shown. These results suggest that dozens of backup servers could be aggregated to one medium-sized remote repository. As future work, we would like to increase the scaling tests.

7.2 Stream-Informed Cache Limitations

Since we do not have a full sketch index, loss of cache locality translates to a loss in potential compression. While earlier experiments showed that stream-informed caching is effective, those experiments were on individual datasets. In a realistic environment, multiple datasets have to share a cache, and garbage collection further degrades locality on disk because live chunks from different containers and datasets can be merged into new containers.

We ran an experiment with a midsize storage appliance with a 288 MB cache sized to handle approximately 20 replicating datasets. The experiment consisted of replicating a real dataset to this appliance while varying the number of synthetic datasets also replicated between 0, 24, and 49. This test was performed with three real datasets. The synthetic datasets were generated with an internal tool that had deduplication of 12X and delta compression of 1.7X, which exercises our caching infrastructure in a realistic manner. When the number of datasets was increased to 25 (1 real and 24 synthetic), delta compression decreased 0%, 6% and 12% among the three real datasets relative to a baseline of replicating each real dataset individually. Increasing to 49 synthetic datasets (beyond what is advised for this hardware) caused delta compression to decrease 0%, 12%, and 27% from the baseline for the three real datasets. Our intuition is that the variability in results is due to locality differences among these datasets. In general, these results suggest our caching technique degrades in a gradual manner as the number of replicating datasets increases relative to the cache size.

This experiment investigates how multiple datasets sharing a cache affect delta compression, and we validate these findings with results from the field presented in Section 8, where customers achieved 2X additional delta compression beyond deduplication even though their systems had multiple datasets sharing a storage appliance. While we do not know the upper bound on how much delta compression these customers could have achieved in a single-dataset scenario, these results suggest sizable network savings.

8 Results from Customers

Basic replication has been available with EMC Data Domain systems for many years using the deduplication protocol of Figure 3, and the extra delta compression stage became available in 2009. The version available to customers has a cache scaled to the number of supported replication streams.

We analyzed daily reports from several hundred storage systems used by our customers during the second week of August 2011, including a variety of hardware configurations. Reporting median values, a typical customer transferred 1 TB of data across a 3.1 Mb/s link during the week, though because of our compression techniques, much less data was physically transferred across the network. Median total compression was 32*X* including deduplication, delta, and local compression. Figure 10 shows the distribution of delta compression with 50% of customers achieving over 2*X* additional compression beyond what deduplication alone achieves, and outliers achieving 5*X* additional delta compression. Concurrent work [38] provides further analysis of replication and backup storage in general.

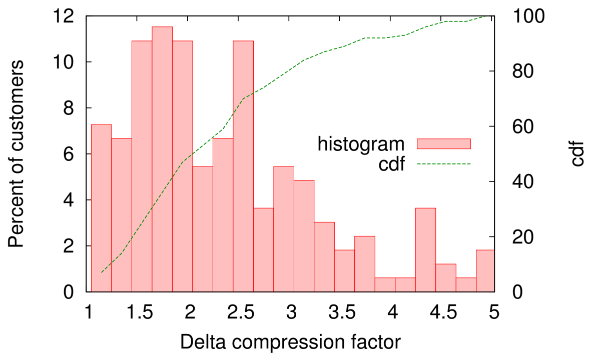


Figure 10: Distribution of delta compression. 50% of customers achieve over 2X additional delta compression.

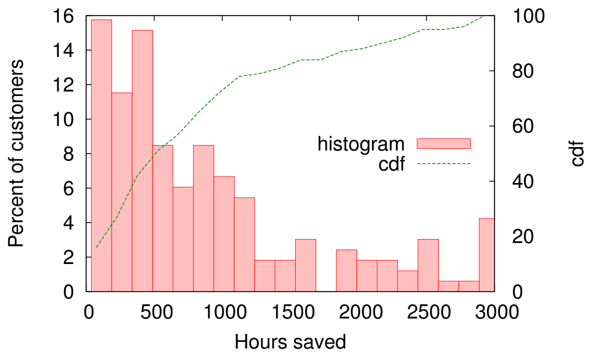


Figure 11: Distribution of hours saved by customers. We estimate that 50% of customers save over 588 hours of replication time per week because of our combination of compression techniques.

Finally, in Figure 11, we show how much time was saved by our customers versus sending data without any compression. Our reports indicate how much data was transferred, an estimate of network throughput (though periodic throttling is difficult to extract), and compression, so we can calculate how long replication would take without compression. The median customer would need 608 hours to fully replicate their data (more hours than are in a week), but with our combined compression, replication reduced to 20 hours (saving 588 hours of network transfer time). For such customers, it would be impossible for them to replicate their data each week without compression, so delta replication significantly increases the amount of data that can be protected.

9 Related Work

Our stream-informed delta replication project builds upon previous work in the areas of optimizing network transfer, delta compression, similarity detection, deduplication, and caching techniques.

Minimizing network transfer has been an area of ongoing research. One of the earliest projects by Spring et al. [33] removed duplicate regions in packets with a synchronized cache by expanding from duplicate starting points. LBFS [25] divided a client's file into chunks and deduplicated chunks against any previously stored. Jumbo Store [14] used a hierarchical representation of files that allowed them to quickly check whether large subregions of files were unchanged. CZIP [26] applied a similar technique with user level caches to remove duplicate chunks while synchronizing remote repositories.

Most work in file synchronization has assumed that versions are well identified so that compression can be achieved relative to one (or a few) specified file(s). Rsync [37] is a widely used tool for synchronizing folders of files based on compressing against files with the same pathname. An improvement [35] recursively split files to find large duplicate regions using a memory cache.

Beyond finding duplicates during network transfer, delta compression is a well known technique for computing the difference between two files or data objects [17, 36]. Delta compression was applied to web pages [8, 24] and file transfer and storage [7, 9, 21, 34] using a URL and file name, respectively, to identify a previous version.

When versioning information is unavailable, a mechanism is needed to find a previous, similar file or data object to use as the base for delta compression. Broder [4, 5] performed some of the early work in the resemblance field by creating features (such as Rabin fingerprints [31]) to represent data such that similar data tend to have identical features. Features were further grouped into super-features to improve matching efficiency by reducing the number of indexes. Features and super-features were used to select an appropriate base file for deduplication and delta compression [12, 19], removing the earlier requirement for versioning information. TAPER [18] presented an alternative to super-features by representing files with a Bloom filter storing chunk fingerprints and measuring file similarity based on the number of matching bits between Bloom filters and then delta compressing similar files. Delta compression within the storage system has used super-feature techniques to identify similar files or regions of files [1, 40]. Aronovich et al. [1] used 16 MB chunks to decrease sketch indexing requirements and had hundreds of disk spindles for performance.

Storage systems have eliminated duplicate regions based on querying an index of fingerprints [3, 22, 29, 30]. Noting that the fingerprint index becomes much larger than will fit in memory and that disk accesses can become the bottleneck, Zhu et al. [41] presented a technique to take advantage of stream locality to reduce disk accesses by 99%. Several variants of this approach explored alternative indexing strategies to load a fingerprint cache such as moving the index to flash memory [10] and indexing a subset of fingerprints either based on logical or post-deduplication layout on disk [2, 16, 20, 23, 39]. Our similarity detection approach builds upon these caching ideas to load sketches as well as fingerprints into a stream-informed cache.

10 Conclusion and Future Work

In this paper, we present stream-informed delta compression for replication of backup datasets across a WAN. Our approach leverages deduplication locality to also find similarity matches used for delta compression. While locality properties of duplicate data have been previously studied, we present the first evidence that similar data has the same locality. We show that using a compact stream-informed cache to load sketches achieves almost as much delta compression as using a full index without extra data structures. Our technique has been incorporated into the Data Domain systems, and average customers achieve 2X additional compression beyond deduplication and save hundreds of hours of replication time each week.

In future work, we would like to expand the number of WAN environments that benefit from delta replication by improving the read throughput, which currently gates our system. Also, we would like to further explore delta compression techniques to improve compression and scalability.

Acknowledgments

We thank Fred Douglis, Kai Li, Stephen Manley, Hugo Patterson, Hyong Shim, Benjamin Zhu, Cezary Dubnicki (our shepherd), and our reviewers for their feedback. We would also like to acknowledge the many EMC engineers who continue to improve and support delta replication.

References

[1] L. Aronovich, R. Asher, E. Bachmat, H. Bitner, M. Hirsch, and S. T. Klein. The design of a similarity based deduplication system. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, pages 6:1–6:14, New York, NY, USA, 2009.

[2] D. Bhagwat, K. Eshghi, D. D. Long, and M. Lillibridge. Extreme binning: scalable, parallel deduplication for chunk-based file backup. In *Proceedings of the 17th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, Sept. 2009.

[3] D. R. Bobbarjung, S. Jagannathan, and C. Dubnicki. Improving duplicate elimination in storage systems. *Trans. Storage*, 2:424–448, November 2006.

[4] A. Broder. On the resemblance and containment of documents. In *Proceedings of the Compression and Complexity of Sequences*, page 21, 1997.

[5] A. Broder. Identifying and filtering near-duplicate documents. In *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, pages 1–10, 2000.

[6] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations (extended abstract). In *Proceedings of the 30th annual ACM symposium on Theory of computing*, pages 327–336, New York, NY, USA, 1998.

[7] R. C. Burns and D. D. E. Long. Efficient distributed backup with delta compression. In *Proceedings of the 5th workshop on I/O in parallel and distributed systems*, pages 27–36, New York, NY, USA, 1997.

[8] M. C. Chan and T. Y. C. Woo. Cache-based compaction: a new technique for optimizing web transfer. In *INFOCOM'99 conference*, March 1999.

[9] Y. Chen, Z. Qu, Z. Zhang, and B.-L. Yeo. Data redundancy and compression methods for a disk-based network backup system. *International Conference on Information Technology: Coding and Computing*, 1:778, 2004.

[10] B. Debnath, S. Sengupta, and J. Li. Chunkstash: speeding up inline storage deduplication using flash memory. In *Proceedings of the USENIX Annual Technical Conference*, Berkeley, CA, USA, 2010.

[11] W. Dong, F. Douglis, K. Li, H. Patterson, S. Reddy, and P. Shilane. Tradeoffs in scalable data routing for deduplication clusters. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, 2011.

[12] F. Douglis and A. Iyengar. Application-specific delta-encoding via resemblance detection. In *Proceedings of the USENIX Annual Technical Conference*, pages 113–126, 2003.

[13] EMC Corporation. Data Domain Boost Software, 2010. `http://www.datadomain.com/products/dd-boost.html`.

[14] K. Eshghi, M. Lillibridge, L. Wilcock, G. Belrose, and R. Hawkes. Jumbo store: Providing efficient incremental upload and versioning for a utility rendering service. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, 2007.

[15] J. L. Gailly and M. Adler. The GZIP compressor. `http://www.gzip.org`.

[16] F. Guo and P. Efstathopoulos. Building a high-performance deduplication system. In *Proceedings of the USENIX Annual Technical Conference*, 2011.

[17] J. J. Hunt, K.-P. Vo, and W. F. Tichy. Delta algorithms: an empirical analysis. *ACM Trans. Softw. Eng. Methodol.*, 7:192–214, April 1998.

[18] N. Jain, M. Dahlin, and R. Tewari. Taper: tiered approach for eliminating redundancy in replica synchronization. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies*, 2005.

[19] P. Kulkarni, F. Douglis, J. LaVoie, and J. M. Tracey. Redundancy elimination within large collections of files. In *Proceedings of the USENIX Annual Technical Conference*, pages 59–72, 2004.

[20] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble. Sparse indexing: large scale, inline deduplication using sampling and locality. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies*, pages 111–123, 2009.

[21] J. MacDonald. File system support for delta compression. Master's thesis, Department of Electrical Engineering and Computer Science, University of California at Berkeley, 2000.

[22] U. Manber. Finding similar files in a large file system. In *Proceedings of the USENIX Winter Technical Conference*, pages 1–10, 1994.

[23] J. Min, D. Yoon, and Y. Won. Efficient deduplication techniques for modern backup operation. *IEEE Transactions on Computers*, 99, 2010.

[24] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for http. In *Proceedings of the ACM SIGCOMM 1997 Conference*, pages 181–194, 1997.

[25] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *SOSP '01: Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 174–187, 2001.

[26] K. Park, S. Ihm, M. Bowman, and V. S. Pai. Supporting practical content-addressable caching with

CZIP compression. In *Proceedings of the USENIX Annual Technical Conference*, pages 14:1–14:14, Berkeley, CA, USA, 2007.

[27] N. Park and D. Lilja. Characterizing datasets for data deduplication in backup applications. In *IEEE International Symposium on Workload Characterization*, 2010.

[28] 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*, Berkeley, CA, USA, 2002.

[29] C. Policroniades and I. Pratt. Alternatives for detecting redundancy in storage systems data. In *Proceedings of the USENIX Annual Technical Conference*, pages 73–86, 2004.

[30] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, 2002.

[31] M. O. Rabin. Fingerprinting by random polynomials. Technical report, Center for Research in Computing Technology, 1981.

[32] Riverbed Technology. Riverbed Steelhead Product Family, 2011. `http://www.riverbed.com/us/assets/media/documents/data_sheets/DataSheet%-Riverbed-FamilyProduct.pdf`.

[33] N. T. Spring and D. Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *Proceedings of the ACM SIGCOMM 2000 Conference*, pages 87–95, 2000.

[34] T. Suel and N. Memon. Algorithms for delta compression and remote file synchronization. In K. Sayood, editor, *Lossless Compression Handbook*. 2002.

[35] T. Suel, P. Noel, and D. Trendafilov. Improved file synchronization techniques for maintaining large replicated collections over slow networks. In *20th International Conference on Data Engineering*, 2004.

[36] D. Trendafilov, N. Memon, and T. Suel. zdelta: An efficient delta compression tool. Technical report, Department of Computer and Information Science at Polytechnic University, 2002.

[37] A. Tridgell. *Efficient algorithms for sorting and synchronization*. PhD thesis, Australian National University, April 2000.

[38] 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 USENIX Conference on File and Storage Technologies*, 2012.

[39] W. Xia, H. Jiang, D. Feng, and Y. Hua. Silo: A similarity-locality based near-exact deduplication scheme with low ram overhead and high throughput. In *Proceedings of the USENIX Annual Technical Conference*, 2011.

[40] L. You and C. Karamanolis. Evaluation of efficient archival storage techniques. In *Proceedings of the 21st Symposium on Mass Storage Systems*, Apr. 2004.

[41] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 269–282, February 2008.

Power Consumption in Enterprise-Scale Backup Storage Systems

Zhichao Li† Kevin M. Greenan‡ Andrew W. Leung‡ Erez Zadok†
†*Stony Brook University* ‡*Backup Recovery Systems Division*
EMC Corporation

Abstract

Power consumption has become an important factor in modern storage system design. Power efficiency is particularly beneficial in disk-based backup systems that store mostly cold data, have significant idle periods, and must compete with the operational costs of tape-based backup. There are no prior published studies on power consumption in these systems, leaving researchers and practitioners to rely on existing assumptions. In this paper we present the first analysis of power consumption in real-world, enterprise, disk-based backup storage systems. We uncovered several important observations, including some that challenge conventional wisdom. We discuss their impact on future power-efficient designs.

1 Introduction

Power has become an important design consideration for modern storage systems as data centers now account for close to 1.5% of the world's total energy consumption [14], with studies showing that up to 40% of that power comes from storage [25]. Power consumption is particularly important for disk-based backup systems because: (1) they contain large amounts of data, often storing several copies of data in higher storage tiers; (2) most of the data is cold, as backups are generally only accessed when there is a failure in a higher storage tier; (3) backup workloads are periodic, often leaving long idle periods that lend themselves to low power modes [31, 35]; and (4) they must compete with the operational costs of low power, tape-based backup systems.

Even though there has been a significant amount of work to improve power consumption in backup or archival storage systems [8, 21, 27], as well as in primary storage systems [3, 33, 36], there are no previously published studies of how these systems consume power in the real world. As a result, power management in backup storage systems is often based on assumptions and commonly held beliefs that may not hold true in practice. For example, prior power calculations have assumed that the only power needed for a drive is quoted in the vendor's specification sheet [8, 27, 34]. However, an infrastructure, including HBAs, enclosures, and fans, is required to support these drives; these draw a non-trivial amount of power, which grows proportionally with the number of drives in the system.

In this paper, we present the first study of power consumption in real-world, large-scale, enterprise, disk-based backup storage systems. We measured systems representing several different generations of production hardware using various backup workloads and power management techniques. Some of our key observations include considerable power consumption variations across seemingly similar platforms, disk enclosures that require more power than the drives they house, and the need for many disks to be in a low-power mode before significant power can be saved. We discuss the impact of our observations and hope they can aid both the storage industry and research communities in future development of power management technologies.

2 Related Work

Empirical power consumption studies have guided the design of many systems outside of storage. Mobile phones and laptop power designs, which are both sensitive to battery lifetime, were influenced by several studies [7, 17, 22, 24]. In data centers, studies have focused on measuring CPU [18, 23], OS [5, 6, 11], and infrastructure power consumption [4] to give an overview of where power is going and the impact various techniques have, such as dynamic voltage and frequency scaling (DVFS). Recently, Sehgal et al. [26] measured how various file system configurations impact power consumption.

Existing storage system power management has largely focused on managing disk power consumption. Much of this existing work assumes that as storage systems scale their capacity—particularly backup and archival systems—the number of disks will increase to the point where disks are the dominant power consumers. As a result, most solutions try to keep as many drives powered-off as possible, spun-down, or spun at a lower RPM. For example, archival systems like MAID [8] and Pergamum [27] use data placement, scrubbing, and recovery techniques that enable many of the drives in the system to be in a low-power mode. Similarly, PARAID [33] allows transitioning between several different RAID layouts to trade-off energy, performance, and reliability. Hibernator [36] allows drives in a RAID array to operate at various RPMs, reducing power consumption while limiting the impact to performance. Write Off-Loading [19] redirects writes from low-power disks to available storage elsewhere, allowing disks to stay in a low-power mode longer.

Our goal is to provide power consumption measurements from real-world, enterprise-scale backup systems, to help guide designs of power-managed storage systems.

3 Methodology

We measured several real-world, enterprise-class backup storage systems. Each used a Network-Attached-Storage (NAS) architecture with a storage controller connected to multiple, external disk drive enclosures. Figure 1 shows the basic system architecture. Each storage controller exports to file-based interfaces to clients, such as NFS and CIFS—and backup-based interfaces, such as VTL and those of backup software (e.g., Symantec's OST [20]). Each storage controller performs inline data deduplication; typically these systems contain more CPUs and memory than other storage systems to perform chunking and to maintain a chunk index.

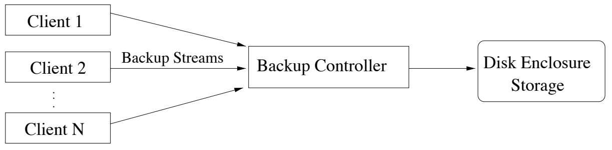


Figure 1: Backup system architecture

	DD880	DD670	DD860	DDTBD
Ship Year	2009	2010	2011	Future
Intel CPU	X7350	E5504	E5504	E7-4870
# CPUs	2	1	2	4
RAM	64GB	16GB	72GB	256GB
NVRAM	2GB	1GB	1GB	4GB
# Disks	4	7	4	4
# Pow Sup	2	2	2	4
# Fans	8	8	8	8
# NICs	1	1	1	2
# HBAs	3	1	3	4

Table 1: Controller hardware summary

Table 1 details the four different EMC controllers that we measured. Each controller was shipped or will be shipped in a different year and represents hardware upgrades over time. Each controller, except for DD670, stores all backup data on disks in external enclosures, and the four disks (three active plus a spare) in the controller store only system and configuration data. DD670 is a low-end, low-cost system that stores both user and system data in its seven disks (six active plus a spare). DDTBD is planned for a future release and does not yet have a model number. Each controller ran the same software version of the DDOS operating system.

Table 2 shows the two different enclosures that we measured. Each enclosure can support various capacity SATA drives. Based on vendor specifications, the drives we used have power usage of about 6–8W idle, 8–12W active, and less than 1W when spun-down. Controllers communicate with the enclosures via Serial Attached SCSI (SAS). Large system configurations can support more than fifty enclosures attached to a single controller, which can host more than a petabyte of physical capacity and tens of petabytes of logical, deduplicated capacity.

	ES20	ES30
Ship Year	2006	2011
# Disks	16	15
# SAS Controllers	2	2
# Power Supplies	2	2
# Fans	2	4

Table 2: Enclosure hardware summary

Experimental setup. We measured controller power consumption using a Fluke 345 Power Quality Clamp Meter [10], an in-line meter that measures the power draw of a device. The meter provides readings with an error of $\pm 2.5\%$. We measured enclosure power consumption using a WattsUP Pro ES [32], another in-line meter, with an accuracy of $\pm 1.5\%$ for measured value plus a constant error of ± 0.3 watt-hours. All measurements were done within a data center environment with room temperature held between 70 °F and 72 °F.

We connected the controllers and enclosures to the meters separately, to measure their power. Thus we present component's measurement separately, rather than as an entire system (e.g., a controller attached to several enclosures). The meters we used allowed us to measure only entire device power consumption, not individual components (e.g., each CPU or HBA) or data-center factors (e.g., cooling or network infrastructure). We present all measurements in watts and all results are an average of several readings with standard deviations less than 5%.

Benchmarks. For each controller and enclosure, we measured the power consumption when idle and when under several backup workloads. Each workload is a standard, reproducible workload used internally to test system performance and functionality. The workloads consist of two clients connecting over a 10 GigE network to a controller writing 36 backup streams. Each backup stream is periodic in nature, where a full backup image is copied to the controller, followed by several incremental backups, followed by another full backup, and so on. For each workload we ran 42 full backup generations. The workloads are designed to mimic those seen in the field for various backup protocols.

	WL-A	WL-B	WL-C
Protocol	NFS	OST	BOOST
Chunking	Server	Server	Client

Table 3: Backup workloads used

We used the three backup protocols shown in Table 3. Clients send backup streams over NFS in WL-A, and over Symantec's OST in WL-B. In both cases, all deduplication is performed on the server. WL-C uses, BOOST [9], an EMC backup client that performs stream chunking on the client side and sends only unique chunks to the server, reducing network and server load. To measure the power consumption of a fully utilized disk subsystem, we used an internal tool that saturates each disk.

4 Discussion

We present our analysis for a variety of configurations in three parts: isolated controller measurements, isolated enclosure measurements, and whole-system analysis using controller and enclosure measurements.

4.1 Controller Measurements

We measured storage controller power consumption under three different scenarios: idle, loaded, and power managed using processor-specific power-saving states.

Controller idle power. A storage controller is considered idle when it is fully powered on, but is not handling a backup or restore workload. In our experiments, each controller was running a full, freshly installed, DDOS software stack, which included several small background daemon processes. However, as no user data was placed on the systems, background jobs such as garbage collection, were not run. Idle power consumption indicates the minimum amount of power a non-power-managed controller would consume when sitting in the data center.

It is commonly assumed that disks are the main contributor to power in a storage system. As shown in Table 4, the controllers can also consume a large amount of power. In the case of DDTBD, the power consumption is almost equal to that of 100 2TB drives [13]. This is significant because even a controller with no usable disk storage can consume a lot of power. Yet, the performance of the controller is critical to maintain high deduplication ratios, and necessary to support petabytes of storage—requiring multiple fast CPUs and lots of RAM. These high idle power-consumption levels are well known [15]. Although computer component vendors have been reducing power consumption in newer systems, there is a long way to go to support true power proportionality in computing systems; therefore, current idle controller power levels must be factored into future designs.

■ **Observation 1:** *The idle controller power consumption is still significant.*

Table 4 shows a large difference in power consumption between controllers. DDTBD consumes almost 3.5× more power than DD670. Here, difference is largely due to the different hardware profiles. DDTBD is a more powerful, high-end controller with significantly more CPU and memory, whereas DD670 is a low-end model. However, this is not the case for the power differences between DD880 and DD860. DD880 consumes more than twice the power as DD860, yet Table 1 shows that their hardware profiles are fairly similar. The amount of CPU and memory plays a major role in power consumption; however, other factors such as the power efficiency of individual components also contribute. Unfortunately, our measurement methodology prevented us from identifying the internal components that contribute to this difference. However, part of this difference can be attributed to DD860 being a newer model with hardware components that consume less power than older models.

	DD880	DD670	DD860	DDTBD
Idle Power (W)	555	225	261	778

Table 4: Idle power consumptions for storage controllers

To better compare controller power consumption, we normalized the power consumption numbers in Table 4 to the maximum usable physical storage capacity. The maximum capacities for the DD880, DD670, DD860, and DDTBD are 192TB, 76TB, 192TB, and 1152TB, respectively. This gives normalized power consumption values of 2.89W/TB for DD880, 2.96W/TB for DD670, 1.35W/TB for DD860, and 0.675W/TB for DDTBD. Although the normalized values are roughly the same for DD880 and DD670, the watts consumed per raw byte trends down with newer generation platforms.

■ **Observation 2:** *Whereas idle controller power consumption varies between models, normalized watts per byte goes down with newer generations.*

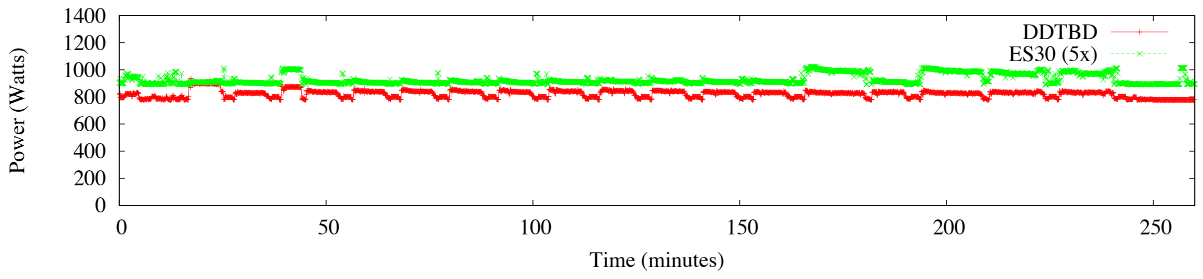
Controller under load. We measured the power consumption of each controller while running the aforementioned workloads. Each controller ran the DDFS deduplicating file system [35] and all required software services. Services such as replication were disabled. The power consumed under load approximates the power typically seen for controllers in-use in a data center. The workloads used are performance-qualification tests that are designed to mimic real customer workloads, but do not guarantee that the controllers are stressed maximally.

Figure 2(a) shows the power consumed by DDTBD while running the WL-A workload. The maximum power consumed during the run was 937W, which is 20% higher than the idle power consumption. Since the power only increased 20% when under load, it may be more beneficial to improve idle consumption before trying to improve active (under load) consumption.

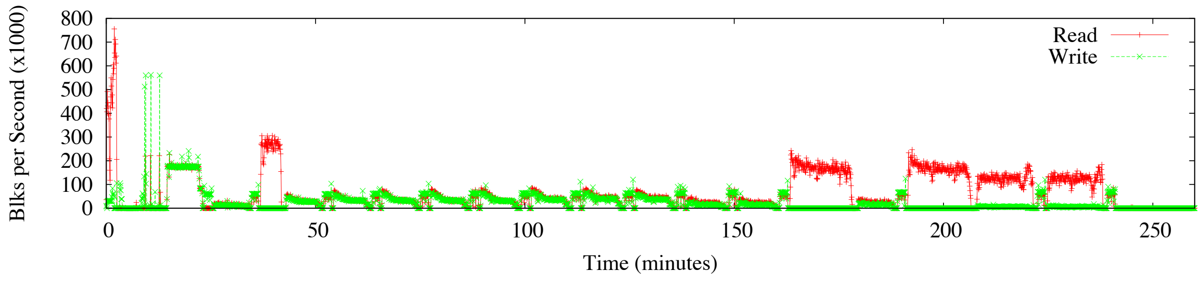
	DD880	DD670	DD860	DDTBD
WL-A	44%	24%	58%	20%
WL-B	58%	29%	61%	36%
WL-C	56%	28%	57%	23%

Table 5: Power increase ratios from idle to loaded system

Table 5 shows the power increase percents from idle to loaded across controller and workload combinations. Several combinations have an increase of less than 30%, while others exceed 50%. Unfortunately, our methodology did not allow us to identify which internal components caused the increase. One noticeable trend is that the increase in power is mostly due to the controller model rather than the workload, as DD880 and DD860 always increased more than DD670 and DDTBD.



(a) Power consumption



(b) I/O statistics

Figure 2: Power consumption and I/O statistics for WL-A on DDTBD, along with the 5 ES30 enclosures attached to it

■ **Observation 3:** *The increase in controller power consumption under load varies much across models.*

I/O statistics from the disk sub-system help explain the increases in controller power consumption. Figure 2(b) shows the number of blocks per second read and written to the enclosures attached to DDTBD during WL-A. We see that a higher rate of disk I/O activity generally corresponds to higher power consumption in both the controller and disk enclosures. Whereas I/Os require the controller to wait on the disk sub-system, they also increase memory copying activity, communication with the subsystem, and deduplication fingerprint hashing.

Power-managed controller. Our backup systems perform in-line, chunk-based deduplication, requiring significant CPU and RAM amounts to compute and manage hashes. As the data path is highly CPU-intensive, applying DVFS techniques during backup—a common way to manage CPU power consumption—can degrade performance. Although it is difficult to throttle CPU during a backup, the backup processes are usually separated by large idle periods, which provide an opportunity to exploit DVFS an other power-saving techniques.

Intel has introduced a small set of CPU power-saving states, which represent a range of CPU states from fully active to mostly powered-off. For example, on the Corei7, C1 uses clock-gating to reduce processor activity, C3 powers down L2 caches, and C6 shuts off the core's power supply entirely [28]. To evaluate the efficacy of the Intel C states on an idle controller, we measured the power savings of the deepest C state. Unfortunately, DDTBD was the only model that supported the Intel C states. We used a modified version of CPUIDLE to place DDTBD into the C6 state [16]. In this state, DDTBD saved just 60W, a mere 8% of total controller power consumption. This finding suggests that DVFS alone is insufficient for saving power in controllers with today's CPUs and a great deal of RAM. Moreover, deeper C states incur higher latency penalties and slow controller performance. We found that the latencies made the controller virtually unusable when in the deepest C state.

■ **Observation 4:** *Placing today's Intel CPUs into deep C state saves only a small amount of power and significantly harms controller performance.*

4.2 Enclosure Measurements

We now analyze the power consumption of two generations of disk enclosures. Similar to Section 4.1, we analyzed the power consumption of the enclosures when idle, under load, and using power-saving techniques.

Enclosure idle power. An enclosure is idle when it is powered on and has no workload running. The idle power consumption of an enclosure represents the lowest amount of power a single enclosure and the housed disks consume without power-management support. Figure 3 shows that an idle ES20 consumes 278W. The number of active enclosures in a high-capacity system can exceed 50, so the total power consumption of the disk enclosures alone can exceed 13kW.

We found that the enclosures have very different power profiles. The idle ES20 consumes 278W, which is 55%

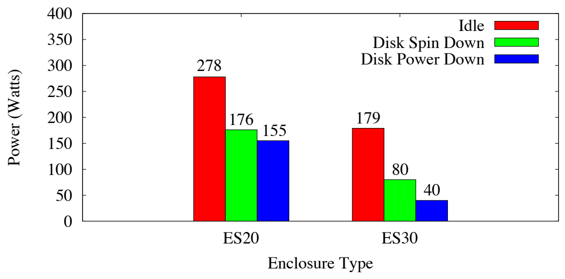


Figure 3: Disk power down vs. spin down. ES20 and ES30 are specified as in Table 2.

higher than the idle ES30, at 179W. We believe that newer hardware largely accounts for this difference. For example, it is well known that power supplies are not 100% efficient. Modern power supplies often place guarantees on efficiency. One standard [1] provides an 80% efficiency guarantee, which means the efficiency will never go below 80% (e.g., for every 10W drawn from the wall, at least 8W is usable by components attached to the power supply). The ES30 has newly designed power supplies, temperature-based fan speeds, and a newer internal controller, which contribute to this difference.

■ **Observation 5:** *The idle power consumption varies greatly across enclosures with new ones being more power efficient.*

Enclosure under load. We also measured the power consumption of each enclosure under the workloads discussed in Section 3. We considered an enclosure under load when it was actively handling an I/O workload.

As shown in Figure 2(a), the total power consumption of the five ES30 enclosures connected to DDTBD, processing WL-A, increased by 10% from 900W when idle to about 1kW. Not surprisingly, Figure 2(b) shows that an increase in enclosure power correlates with an increase in I/O traffic. Percentages for the other enclosure and workload combinations ranged from 6–22%.

Our deduplicating file system greatly reduces the amount of I/O traffic seen by the disk sub-system. As described in Section 3, we used an internal tool to measure the power consumption of a fully utilized disk subsystem. Table 6 shows that ES20 consumption grew by 22% from 278W when idle to 340W. ES30 increased 15% from 179W idle to 205W. Interestingly, these increases are much smaller than those observed for the controllers under load in Section 4.1.

■ **Observation 6:** *The consumption of the enclosures increases between 15% and 22% under heavy load.*

Power managed enclosure. We compared the power consumption of ES20 and ES30 using two disk power-saving techniques: power-down and spin-down. With spin-down, the disk is powered on, but the head is parked and the motor is stopped. With power-down, the enclosure's disk slot is powered off, cutting off all drive power.

	ES20	**ES30**
Idle Power (W)	278	179
Max Power (W)	340	205

Table 6: Max power for enclosures ES20 and ES30

As shown in Figure 3, the relative power savings of the ES20 and ES30 are quite different. For ES30, spin-down reduced power consumption by 55% from 179W to 80W. For ES20, the power dropped by 37% from 278W to 176W. Although the absolute spin-down savings was roughly 100W for both enclosures, power-down was much more effective for ES30 than ES20. Power-down for ES30 reduced power consumption by 78%, but only 44% for ES20. As mentioned in Section 3, each disk consumes less than 1W when spun-down. However, for both ES20 and ES30, power-down saved more than 1W per disk compared to spin-down.

■ **Observation 7:** *Disk power-down may be more effective than disk spin-down for both ES20 and ES30.*

Looking closer at the ES20 power savings, the enclosure actually consumes more power than the disks it is housing (an improvement opportunity for enclosure manufactures). With all disks powered down, ES20 consumes 155W, which is more than the 123W saved by powering down the disks (consistent with disk vendor specs).

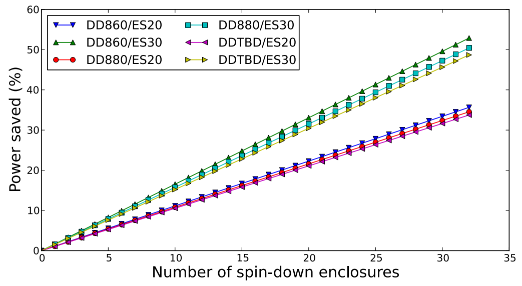
■ **Observation 8:** *Disk enclosures may consume more power than the drives they house. As a result, effective power management of the storage subsystem may require more than just disk-based power-management.*

We observed that an idle ES30 enclosure consumes 64% of an idle ES20, while a ES30 in power-down mode consumes only 25% of the power of an ES20 in power-down mode. This suggests that newer hardware's idle and especially power-managed modes are getting better.

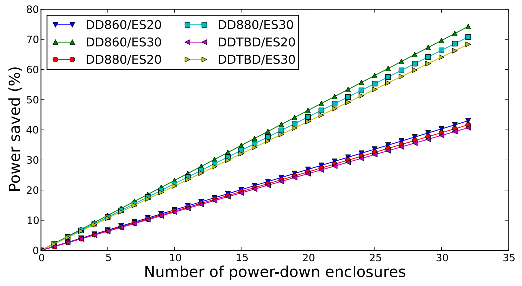
4.3 System-Level Measurements

A common metric for evaluating a power management technique is the percentage of total system power that is saved. We measured the amount of power savings for different controller and enclosure combinations using spin-down and power-down techniques. We considered system configurations with an idle controller and 32 idle enclosures (which totals 512 disks for ES20 and 480 disks for ES30) and we varied the number of enclosures that have all their disks power managed. We excluded DD670 because it supports only up to 4 external shelves.

Figure 4 shows the percentage of total system power saved as the number of enclosures with power-managed disks was increased. In Figure 4(a) disks were spun down, while in Figure 4(b) disks were powered down. We found that it took a considerable number of power-managed disks to yield a significant system power savings. In the best case with DD860 and ES30, 13 of the 32



(a) Disk Spin Down vs. Power Savings Percentage



(b) Disk Power Down vs. Power Savings Percentage

Figure 4: Total system power savings using disk power management

enclosures must have their disks spun down to achieve a 20% power savings. In other words, over 40% of the disks must be spun down to save 20% of the total power. In the worse case with DDTBD and ES20, 19 of the 32 enclosures must have their disks spun down to achieve a 20% savings. This scenario required almost 60% of the disks to be spun down to save 20% of the power. Only two of our six configurations were able to achieve more than 50% savings even when all disks were spun down. These numbers were improved when power down is used, but a large number of disks was still needed to achieve significant savings.

■ **Observation 9:** *To save a significant amount of power, many drives must be in a low power mode.*

The limited power savings is due in part to the controllers consuming a large amount of power. As seen in Section 4.1, a single controller may consume as much power as 100 disks. Additionally, as shown in Section 4.2, disk enclosures can consume more power than all of the drives they house, and the number of enclosures must scale with the number of drives in the system. These observations indicate that for some systems, even aggressive disk power management may be insufficient to save enough power and that power must be saved elsewhere in the system (e.g., reducing controller and enclosure power consumption, new electronics, etc.).

5 Conclusions

We presented the first study of power consumption in real-world, large-scale, enterprise, disk-based backup storage systems. Although we investigated only a handful of systems, we already uncovered a three interesting observations that may impact the design of future power-efficient backup storage systems.

(1) We found that components other than disks consume a significant amount of power, even at large scales. We observed that both storage controllers and enclosures can consume large amounts of power. For example, DDTBD consumes more power than 100 2TB drives and ES20 consumes more power than the drives it houses. As a result, future power-efficient designs should look beyond disks to target controllers and enclosures as well.

(2) We found a large difference between idle and active power consumption across models. For some models, active power consumption is only 20% higher than idle, while it is up to 60% higher for others. This observation indicates that existing systems are not achieving energy proportionality [2, 4, 12, 29, 30], which states that systems should consume power proportional to the amount of work performed. For some systems, we found a disproportionate amount of power used while idle. As backups often run on particular schedules, these systems may spend a lot of time idle, opening up opportunities to further reduce power consumption.

(3) We discovered large power consumption differences between similar hardware. Despite having similar hardware specifications, we observed that the older DD880 model consumed twice as much idle power as the newer DD860 model. We also saw that an idle ES20 consumed 55% more power than an idle ES30. This suggests that the power profile of an existing system can be improved by retiring old hardware with newer, more efficient hardware. We hope to see continuing improvements from manufacturers of electronics and computer parts.

Future work. To evaluate the steady state power profile of a backup storage system, we plan to measure a system that has been aged and a system with active background tasks. For comparison, we would like to study power use of primary storage systems and clustered storage systems, whose hardware and workloads are different than backup systems. Lastly, we would like to investigate the contribution of individual computer component (e.g., CPUs and RAM) on overall power consumption.

Acknowledgements. We thank the EMC/Data Domain performance team for their help. We also thank Windsor Hsu, our shepherd Jiri Schindler and our anonymous reviewers for their helpful feedback. This work was supported in part by NSF award CCF-0937854.

References

[1] 80 PLUS Certified Power Supplies and Manufacturers. *www.plugloadsolutions.com/80PlusPowerSupplies.aspx*.

[2] H. Amur, J. Cipar, V. Gupta, G. R. Ganger, M. A. Kozuch, and K. Schwan. Robust and flexible power-proportional storage. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, 2010.

[3] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '2009)*, pages 1–14. ACM SIGOPS, October 2009.

[4] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *Computer*, 40:33–37, December 2007.

[5] F. Bellosa. The benefits of event: driven energy accounting in power-sensitive systems. In *Proceedings of the 9th workshop on ACM SIGOPS European workshop*, pages 37–42, 2000.

[6] W.L. Bircher and L.K. John. Complete system power estimation: A trickle-down approach based on performance events. In *Proceedings of the 2007 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 158–168, 2007.

[7] A. Carroll and G. Heiser. An analysis of power consumption in a smartphone. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, Boston, MA, USA, 2010.

[8] D. Colarelli and D. Grunwald. Massive Arrays of Idle Disks for Storage Archives. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–11, 2002.

[9] Data Domain Boost Software, EMC Corporation, 2012. *http://www.datadomain.com/products/dd-boost.html*.

[10] Fluke 345 Power Quality Clamp Meter. *www.fluke.com/fluke/caen/products/categorypqttop.htm*.

[11] D. Grunwald, C. B. Morrey III, P. Levis, M. Neufeld, and K. I. Farkas. Policies for dynamic clock scheduling. In *Proceedings of the 4th Symposium on Operating System Design & Implementation*, San Diego, CA, 2000.

[12] J. Guerra, W. Belluomini, J. Glider, K. Gupta, and H. Pucha. Energy proportionality for storage: Impact and feasibility. *ACM SIGOPS Operating Systems Review*, pages 35 – 39, 2010.

[13] Hitachi Deskstar 7K2000. *www.hitachigst.com/deskstar-7k2000*.

[14] J. G. Koomey. Growth in data center electricity use 2005 to 2010. Technical report, Standord University, 2011. *www.koomey.com*.

[15] R. Kothiyal, V. Tarasov, P. Sehgal, and E. Zadok. Energy and Performance Evaluation of Lossless File Data Compression on Server Systems. In *Proceedings of the Second ACM Israeli Experimental Systems Conference (SYSTOR '09)*, Haifa, Israel, May 2009. ACM.

[16] S. Li and A. Belay. cpuidle — do nothing, efficiently... In *Proceedings of the Linux Symposium*, volume 2, Ottawa, Ontario, Canada, 2007.

[17] J. R. Lorch. A Complete Picture of the Energy Consumption of a Portable Computer. Master's thesis, University of California at Berkeley, 1995. *http://research.microsoft.com/users/lorch/papers/masters.ps*.

[18] A. Miyoshi, C. Lefurgy, E. V. Hensbergen, R. Rajamony, and R. Rajkumar. Critical power slope: understanding the runtime effects of frequency scaling. In *Proceedings of the 16th International Conference on Supercomputing (ICS '02)*, pages 35–44, 2002.

[19] D. Narayanan, A. Donnelly, and A. Rowstron. Write off-loading: practical power management for enterprise storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST 2008)*, 2008.

[20] Symantec OpenStorage, Symantec Corporation, 2012. *http://www.symantec.com/theme.jsp?themeid=openstorage*.

[21] E. Pinheiro and R. Bianchini. Energy Conservation Techniques for Disk Array-Based Servers. In *Proceedings of the 18th International Conference on Supercomputing (ICS 2004)*, pages 68–78, 2004.

[22] A. Sagahyroon. Power consumption breakdown on a modern laptop. In *Proceedings of the 2004 Workshop on Power-Aware Computer Systems*, pages 165–180, Portland, OR, 2004.

[23] A. Sagahyroon. Analysis of dynamic power management on multi-core processors. In *Proceedings of the International Symposium on Circuits and Systems*, pages 1721–1724, 2006.

[24] A. Sagahyroon. Power consumption in handheld computers. In *Proceedings of the IEEE Asia Pacific Conference on Circuits and Systems*, pages 1721–1724, Singapore, 2006.

[25] G. Schulz. Storage industry trends and it infrastructure resource management (irm), 2007. *www.storageio.com/DownloadItems/CMG/MSP_CMG_May03_2007.pdf*.

[26] P. Sehgal, V. Tarasov, and E. Zadok. Evaluating Performance and Energy in File System Server Workloads Extensions. In *FAST'10: Proceedings of the 8th USENIX Conference on File and Storage Technologies*, pages 253–266, San Jose, CA, February 2010. USENIX Association.

[27] M. W. Storer, K. M. Greenan, E. L. Miller, and K. Voruganti. Pergamum: replacing tape with energy efficient, reliable, disk-based archival storage. In *Proceedings of the Sixth USENIX Conference on File and Storage Technologies (FAST '08)*, San Jose, CA, February 2008. USENIX Association.

[28] E. L. Sueur and G. Heiser. Slow down or sleep, that is the question. In *Proceedings of the 2011 USENIX Annual Technical Conference*, Portland, Oregon, USA, 2011.

[29] E. Thereska, A. Donnelly, and D. Narayanan. Sierra: a power-proportional, distributed storage system. In *Proceedings of EuroSys 2011*, 2011.

[30] A. Verma, R. Koller, L. Useche, and R. Rangaswami. Srcmap: Energy proportional storage using dynamic consolidation. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, 2010.

[31] 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 Tenth USENIX Conference on File and Storage Technologies (FAST '12)*, San Jose, CA, February 2012. USENIX Association.

[32] Watts up? PRO ES Power Meter. *www.wattsupmeters.com/secure/products.php*.

[33] C. Weddle, M. Oldham, J. Qian, A. A. Wang, P. Reiher, and G. Kuenning. PARAID: A gear-shifting power-aware RAID. In *Proceedings of the Fifth USENIX Conference on File and Storage Technologies (FAST '07)*, pages 245–260, San Jose, CA, February 2007. USENIX Association.

[34] A. Wildani and E. Miller. Semantic data placement for power management in archival storage. In *PDSW 2010*, New Orleans, LA, USA, 2010. ACM.

[35] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the Sixth USENIX Conference on File and Storage Technologies (FAST '08)*, San Jose, California, USA, 2008.

[36] Q. Zhu, Z. Chen, L. Tan, Y. Zhou, K. Keeton, and J. Wilkes. Hibernator: Helping Disk Arrays Sleep Through the Winter. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 177–190, Brighton, UK, October 2005. ACM Press.

Recon: Verifying File System Consistency at Runtime

Daniel Fryer, Kuei Sun, Rahat Mahmood, TingHao Cheng,
Shaun Benjamin, Ashvin Goel, Angela Demke Brown
University of Toronto

Abstract

File system bugs that corrupt file system metadata on disk are insidious. Existing file-system reliability methods, such as checksums, redundancy, or transactional updates, merely ensure that the corruption is reliably preserved. The typical workarounds, based on using backups or repairing the file system, are painfully slow. Worse, the recovery is performed long after the original error occurred and thus may result in further corruption and data loss.

We present a system called Recon that protects file system metadata from buggy file system operations. Our approach leverages modern file systems that provide crash consistency using transactional updates. We define declarative statements called consistency invariants for a file system. These invariants must be satisfied by each transaction being committed to disk to preserve file system integrity. Recon checks these invariants at commit, thereby minimizing the damage caused by buggy file systems.

The major challenges to this approach are specifying invariants and interpreting file system behavior correctly without relying on the file system code. Recon provides a framework for file-system specific metadata interpretation and invariant checking. We show the feasibility of interpreting metadata and writing consistency invariants for the Linux ext3 file system using this framework. Recon can detect random as well as targeted file-system corruption at runtime as effectively as the offline e2fsck file-system checker, with low overhead.

1 Introduction

It is no surprise that file systems have bugs [20, 29, 31]. Modern file systems are designed to support a range of environments, from smart phones to high-end servers, while delivering high performance. Further, they must handle a large number of failure conditions while preserving data integrity. Ironically, the resulting complexity leads to bugs that can be hard to detect even under heavy testing. These bugs can cause silent data corruption [20, 19], random application crashes, or even worse, security exploits [30].

Unlike hardware errors and crash failures, it is much harder to recover from data corruption caused by bugs in file-system code. Hardware errors can be handled by using checksums and redundancy for error detection and recovery [4, 10]. Crash failure recovery can be performed using transactional methods, such as journaling [12], shadow paging [14], and soft updates [9]. Modern file systems, such as ZFS, are carefully designed to handle a wide range of disk faults [32]. However, the machinery used for protecting against disk corruption (e.g., checksums, replication and transactional updates) does not help if the file system code itself is the source of an error, in which case these mechanisms only serve to faithfully preserve the incorrect state.

File system bugs that cause severe metadata corruption appear regularly. We compiled a list of bugs in the Linux ext3 and the recently deployed btrfs file systems, by searching for "ext3 corruption" and "btrfs corruption" in various distribution-specific bug trackers or mailing lists. Based on the bug description and discussions, we removed bugs that did not cause metadata inconsistency, or were not reproducible, or were reported by a single user only. Table 1 summarizes the remaining bugs. Note that ext3, despite its maturity and widespread use, shows continuing reports of corruption bugs. One recent example is not yet closed, while another closed only in 2010 and affected the ext2, ext3 and ext4 file systems. These reports likely under-represent the problem because the bugs that cause metadata corruption may be *fail silent*, i.e., the error is not reported at the time of the original corruption. By the time the inconsistencies appear, the damage may have escalated, making it harder to pinpoint the problem.

When metadata corruption is discovered, it requires complex recovery procedures. Current solutions fall in two categories, both of which are unsatisfactory. One approach is to use disaster recovery methods, such as a backup or a snapshot, but these can cause significant downtime and loss of recent data. Another option is to use an offline consistency check tool (e.g., e2fsck) for restoring file system consistency. While a consistency check can detect most failures, it requires the entire disk to be checked, causing significant downtime for large file systems. This problem is getting worse because disk capacities are growing faster than disk bandwidth and seek time [13]. Furthermore, the consistency check is run after the fact, often after a system crash occurs or even less frequently with journaling file systems. Thus an error may propagate and cause significant damage, making repair a non-trivial process [11]. For example, Section 5 shows that a single byte corruption may cause repair to fail.

To minimize the need for offline recovery methods, our aim is to verify file-system metadata consistency at runtime. Metadata is more vulnerable to corruption by file

FS	Source	Bug Title	Closed
ext3	http://lwn.net/Articles/2663/	ext3 corruption fix	2002-06
ext3	kerneltrap.org/node/515	Linux: Data corrupting ext3 bug in 2.4.20	2002-12
ext3	Redhat, #311301	panic/ext3 fs corruption with RHEL4-U6-re20070927.0	2007-11
ext3	https://lkml.org/lkml/2008/12/6/88	Re: [2.6.27] filesystem (ext3) corruption (access beyond end)	2008-06
ext3	Debian, #425534	linux-2.6: ext3 filesystem corruption	2008-09
ext3	Debian, #533616	linux-image-2.6.29-2-amd64: occasional ext3 filesystem corruption	2009-06
ext3	Redhat, #515529	ENOSPC during fsstress leads to filesystem corruption on ext2, ext3, and ext4	2010-03
ext3	https://lkml.org/lkml/2011/6/16/99	ext3: Fix fs corruption when make_indexed_dir() fails	2011-06
ext3	Redhat, #658391	Data corruption: resume from hibernate always ends up with EXT3 fs errors	Not yet
btrfs	https://lkml.org/lkml/2009/8/21/45	btrfs rb corruption fix	2009-08
btrfs	https://lkml.org/lkml/2010/2/25/376	[2.6.33 regression] btrfs mount causes memory corruption	2010-02
btrfs	https://lkml.org/lkml/2010/11/8/248	DM-CRYPT: Scale to multiple CPUs v3 on 2.6.37-rc* ?	2010-09
btrfs	https://lkml.org/lkml/2011/2/9/172	[PATCH] btrfs: prevent heap corruption in btrfs_ioctl_space_info()	2011-02
btrfs	https://lkml.org/lkml/2011/4/26/304	btrfs updates (slab corruption in btrfs fitrim support)	2011-04

Table 1: File system bugs causing data corruption. All Red Hat and Debian bugs are rated high-severity. The severity level of bugs obtained from mailing lists is not known.

system bugs because the file system directly manipulates the contents of metadata blocks. Metadata corruption may also result in significant loss of user data because a file system operating on incorrect metadata may overwrite existing data or render it inaccessible.

We present a system called Recon that aims to preserve metadata consistency in the face of *arbitrary* file-system bugs. Our approach leverages modern file systems that provide crash consistency using transactional methods, such as journaling [28, 6, 27] and shadow paging file systems [14, 4, 16]. Recon checks that each transaction being committed to disk preserves metadata consistency. We derive the checks, which we call consistency invariants, from the consistency rules used by the offline file system checker. A key challenge is to correctly interpret file system behavior without relying on the file system code. Recon provides a block-layer framework for interpreting file system metadata and invariant checking.

An important benefit of Recon is its ability to convert fail-silent errors into detectable invariant violations, raising the possibility of combining Recon with file system recovery techniques such as Membrane [26], which are unable to handle silent failures.

Our current implementation of Recon shows the feasibility of interpreting metadata and writing consistency invariants for the widely used Linux ext3 file system. Recon checks ext3 invariants corresponding to most of the consistency properties checked by the e2fsck offline check program. It detects random and type-specific file-system corruption as effectively as e2fsck, with low memory and performance overhead. At the same time, our approach does not suffer from the limitations of offline checking described earlier because corruption is detected immediately. The rest of the paper describes our approach in detail and presents the results of our initial evaluation.

2 Approach

The Recon system interposes between the file system and the storage device at the block layer and checks a set of consistency invariants before permitting metadata writes to reach the disk. We derive the invariants from the rules used by the file system checker. As an example, the e2fsck program checks that file system blocks are not doubly allocated. Our invariants check this property at runtime and thus prevent file-system bugs from causing any double allocation corruption on disk.

Figure 1 shows the architecture of the Recon system. Recon provides a framework for caching metadata blocks and an API for checking file-system specific invariants using its metadata cache. A separate cache is maintained because the file system cache is untrusted and because it allows checking the invariants efficiently. Besides ext3, we have also examined the consistency properties of the Linux btrfs file system and implemented several btrfs invariants. The paper describes our initial experience with adapting our system for btrfs.

Our approach addresses three challenges: 1) *when* should the consistency properties be checked, 2) *what* properties should be checked, and 3) *how* should they be checked. Below, we describe these challenges and how we address them. The caching framework and the file-system specific Recon APIs are described in Section 4.

2.1 When to Check Consistency?

The in-memory copies of metadata may be temporarily inconsistent during file system operation and so it is not easy to check consistency properties at arbitrary times. Instead, checks can be performed when the file system itself claims that metadata is consistent. For example, journaling and shadow-paging file systems are already designed

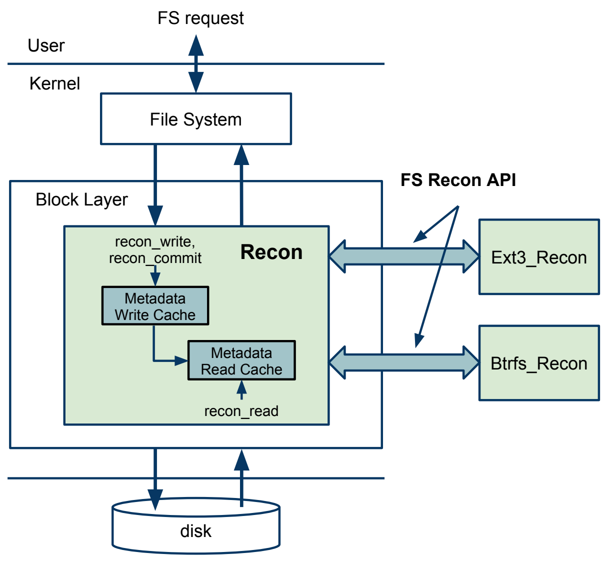


Figure 1: The Recon Architecture

to ensure crash consistency using transactional methods, wherein disk blocks from one or more operations, such as the creation of a directory and a file write, are grouped into transactions. Transaction commits are well-defined points at which the file system believes that it is consistent, and hence transaction boundaries serve as convenient vantage points for verifying consistency properties. Recon checks transactions *before* they commit, thereby ensuring that a committed transaction is consistent, even in the face of arbitrary file system bugs.[1]

Checking consistency for shadow paging systems is relatively straightforward because all transaction data is written to disk before the commit block. For example, btrfs writes all blocks in a transaction, and then commits the transaction by writing its superblock. Recon checks each transaction before the superblock is written to disk.

Checking consistency for journaling file systems is more complicated because transaction data is written to disk both before and after the commit block. For example, ext3 writes metadata to disk in several steps: 1) write metadata to journal, 2) write commit block to journal, at which point the transaction is committed, 3) write (or checkpoint) metadata to its final destination on disk, and 4) free space in the journal.

During Step 1, Recon copies metadata blocks into its write cache, giving it a view of all the updates in a transaction. Then it checks the ext3 transaction in Step 2, i.e., before the commit block is written to the journal, which ensures that all blocks in the transaction are checked for consistency before they become durable. Checking consistency after commit could lead to checkpointing a corrupt block, and furthermore it would not be possible to undo such corruption. Besides checking consistency at commit, we also need to verify the checkpointing process. This step requires checking that all the committed blocks and their contents are checkpointed correctly.

[1]Implementing consistency invariants for soft update file systems [9] that provide consistency after each write but do not use transactions should be possible but will likely be more complicated.

2.2 What Consistency Properties to Check?

Identifying the correct consistency properties is challenging because the behavior of the file system is not formally specified. Fortunately, we can derive an informal specification of metadata consistency properties from offline file-system consistency checkers, such as the Linux e2fsck program. For example, Gunawi et al. found that the Linux e2fsck program checks 121 properties that are common to both ext2 and ext3 file systems and some ext3 journal properties and optional features [11].

These consistency properties define what it means to have consistent metadata on disk. Our aim is to ensure that any metadata committed to disk will maintain these same consistency properties. Unfortunately, consistency properties are *global* statements about the file system. For example, a simple check implemented by e2fsck is that the deletion times of *all* used inodes are zero. Determining the in-use status of all inodes, and checking the deletion time of all used inodes is infeasible at every transaction commit. Similarly, another consistency property is that *all* live data blocks are marked in the block bitmap. Checking these global properties requires a full disk scan.

Instead, we derive a *consistency invariant* from each consistency property. The invariant is a local assertion that must hold for a transaction to preserve the corresponding file system consistency property. For example, consider the "all live data blocks are marked in the block bitmap" property. The corresponding consistency invariant is that a transaction that makes a data block live (i.e., by adding a pointer to the block) must also contain a corresponding bit-flip (from 0 to 1) in the block bitmap within the same transaction, i.e., the invariant is "block pointer set from 0 to N $\Leftrightarrow$ bit N set in bitmap". This invariant can be checked by examining only the updated blocks, i.e., the updated pointer block and the updated block bitmap must be part of the same transaction. We describe this invariant in more detail in Section 3.2.

We structure a consistency invariant as an implication, $A \Rightarrow B$. The premise A *always* involves an update to some data structure field, and hence checking the invariant is triggered by a change in that field. When such an update occurs then the conclusion B must be true to preserve the invariant. If a converse $B \Rightarrow A$ invariant also exists, then we refer to the two invariants as a biconditional invariant $A \Leftrightarrow B$, as shown in the example above.

We rely on the ability to convert consistency properties requiring global information into invariants that can be checked using information "local" to the transaction, as described in the previous example. Such a transformation must be possible because file systems keep themselves consistent without examining the entire disk state. In other words, our invariant checking should not require much more data than the file system itself needs for its operations. Section 5 shows that this is indeed the case because Recon has low overheads.

Finally, our invariant checking approach relies on an inductive argument. It assumes that the file system is consistent before each transaction. If the updates in the transaction meet the consistency invariants, the file system will remain consistent after the transaction. Likewise, if an invariant is violated, there is potential for data loss or incorrect data being returned to applications. Section 2.4 provides more details about our assumptions.

2.3 How to Check Consistency Invariants?

Consistency invariants are expressed in terms of logical file-system data structures, such as current and updated values of block pointers, bits in block bitmap, etc.. However, Recon needs to observe physical blocks below the file system because it cannot trust a buggy file system to provide the correct logical data structure information. We bridge this semantic gap by inferring the types of metadata blocks when they are read or written, which allows parsing and interpreting them, similar to semantically smart disks [24]. Then Recon checks invariants on the typed blocks at commit points, as described below.

Metadata interpretation Block typing and metadata interpretation depend on the idea that file systems access metadata by following a graph of pointers. For example, a pointer to a block is read before the pointed-to block is read, which we call the pointer-before-block assumption. These pointers may be explicit block pointers or are implied by the structure of the file system. For example, ext3 will read an inode containing a pointer to an indirect block before reading the indirect block. When an inode block is read, Recon copies it into its read cache and then parses the inodes in the block to create a mapping from a block to its type for any metadata blocks pointed to by the inodes. In this case, Recon creates a block-type mapping associating the "indirect block" type with the block pointed to by the EXT3_IND_BLOCK pointer in the inode. As a result, Recon recognizes an indirect block when it is read.

Similarly, the block group descriptor (BGD) tables in ext3 describe the locations of inode blocks and inode and block allocation bitmaps. The BGD tables must be read before any of the blocks that they point to, allowing Recon to create block-type mappings for inode and bitmap blocks. This block-type identification is bootstrapped using the superblock, which exists at a known location.

When a metadata block is newly allocated in a transaction, Recon does not yet know its type. In this case, there must exist an updated metadata block in the transaction with a known type that points to this unclassified block directly or indirectly, or else the newly allocated block would not be reachable in the file system. By following the path of pointers from the known metadata block to the newly allocated block, Recon can always create block-type mappings for newly allocated blocks.

For example, suppose a block is allocated to an indirect block of a file. If the file already existed then its inode block must have been read and updated in the transaction. Since the inode block was read previously, Recon knows its type and can determine the type of the newly allocated indirect block. Similarly, if the file did not exist then its parent directory must have existed and been updated, which helps determine the types of the (possibly newly allocated) inode block and then the indirect block. Determining the types of newly allocated blocks may require multiple passes over the blocks updated in the transaction. At the end, all new metadata blocks must be typed or else the pointer-before-block assumption is violated.

Commit processing At commit, Recon uses the block-type mapping to determine the data structures in each of the (updated) transaction blocks, available in the Recon write cache. These data structures are compared with their previous versions, which are derived from the Recon read cache, at the granularity of data structure fields. Each field update generates a logical *change record* with the format [*type, id, field, oldval, newval*].

The *type* specifies a data structure (e.g., inode, directory block). The *id* is the unique identifier of a specific object of the given type (e.g. inode number). The (type, id) pair allows locating the specific data structure in the file system image. The *field* is a field in the structure (e.g. inode size field) or a key from a set (e.g. directory entry name). The *oldval* and *newval* are the old and new values of the corresponding field. These records are generated for existing, newly allocated and deallocated metadata blocks. When an item is newly created or allocated, the oldval is ϕ (a sentinel value). Similarly, when an item is destroyed or deallocated, the newval is ϕ.

Figure 2 shows an example of a set of change records associated with an ext3 transaction in which a single write operation increases the size of a file from one block to two blocks. Change records serve as an abstraction, cleanly separating the interpretation of physical metadata blocks from invariant checking on logical data structures. We show how invariants are implemented using change records in Section 3. When all invariants are checked successfully, the transaction is allowed to commit.

2.4 Fault Model

Our goal is to preserve file-system metadata consistency in the presence of arbitrary file-system bugs. We make

```
[Inode, 12, block[1], 0, 22717]      ; In inode 12, direct block ptr 1 is set to block 22717
[BBM, 22717, 0, 0, 1]                ; Block 22717 is marked allocated in block bitmap
[BGD, 0, free_blocks, 1500, 1499]    ; In block group 0, nr. of free blocks decreases by 1
[Inode, 12, i_size, 4010, 7249]      ; i_size field increases from 4010 to 7249 bytes
[Inode, 12, i_blocks, 8, 16]         ; i_blocks is the number of sectors used by file
[Inode, 12, mtime, 1-18-12, 1-20-12] ; timestamp change
[Inode, 12, ctime, 1-16-12, 1-20-12] ; timestamp change
```

Figure 2: Change records when a block is added to a file

three assumptions to provide this guarantee. First, we assume that the Recon code and its invariant checks are correct and immutable and the Recon metadata cache is protected. If these assumptions are incorrect, it is unlikely that an inconsistent transaction would pass our checks, because the file-system bug and our corrupted check would need to be correlated. However, Recon may generate false alarms, indicating corruption even when a transaction is consistent. Such corruption is still an indication of a bug in the overall system. A hypervisor-based Recon implementation would provide stronger isolation of the Recon code and data from the kernel, helping ensure metadata consistency in the face of *arbitrary kernel* bugs.

Second, if the ext3 file system writes a metadata block before Recon knows its type then Recon will assume that a data block is being written and will allow the operation. For example, a file system bug may corrupt the block number in a disk request structure and cause a misdirected write to a metadata block. Recon will not detect this error because the write violates our pointer-before-block assumption, and ext3 does not provide any other way to identify the block being updated.[2] As future work, we plan to retrofit ext3 to allow such identification. Misdirected writes will not cause a problem with btrfs because its extents are self-identifying [2].

Finally, our inductive assumption about metadata consistency before each transaction (discussed in Section 2.2) requires correct functioning of the lower layers of the system, including the Linux block device layer and all hardware in the data path. It is possible to detect and recover from errors at these layers by using metadata checksums and redundancy. This functionality could be implemented at the block layer for the ext3 file system [10]. The btrfs file system already provides such functionality [16]. If these assumptions are not met, offline checking and repair should be used as a last resort.

3 Consistency Invariants

A file system checker verifies file system consistency by applying a comprehensive set of rules for detecting and optionally repairing inconsistencies. We are primarily interested in checking consistency properties and can reuse the rules associated with detecting, but not repairing, inconsistencies. We have applied our approach to the ext3 and the btrfs file systems. Below, we provide an overview of the consistency rules for these file systems.

The SQCK system [11] encapsulates the 121 checks of the ext3 fsck program in a set of SQL queries. Although there is a close correspondence between SQCK queries and e2fsck checks, some SQCK queries combine multiple checks. Table 2 provides a breakdown of the number of rules checked by SQCK for different file-system data structures. We show 101 rules in Table 2, because the rest are used for repair. The simplest checks (lines starting with the word *Within*) examine individual structures (e.g., superblock fields, inode fields, and directory entries appear valid). Some checks ensure that pointers lie within an expected range. More complicated checks (lines starting with the word *Between*) ensure that block pointers (across all files) do not point to the same data blocks, and directories form a connected tree.

We have done a similar classification of the rules checked by the btrfs checker, as shown in Table 3. Btrfs is an extent-based, B-tree file system that stores file-system metadata structures (e.g., inodes, directories, etc.) in B-tree leaves [16]. It uses a shadow-paging transaction model for updates and for supporting file-system snapshots. Extent allocation information is maintained in an extent B-tree, which serves the same purpose as ext3 block bitmaps. The roots for all the B-trees are maintained in a top-level B-tree called the root tree. Although the btrfs checker is still a work in progress (e.g., it performs no repair), currently it uses 30 rules for detecting inconsistencies. Of these, the first four rule sets are used to check the structure of the B-tree, while the rest deal with typical file-system objects such as inodes and directories.

Next, we provide several examples that show how we transform the consistency properties for various data structures shown in Tables 2 and 3 into invariants. An invariant is implemented by pattern matching change records. When such a match occurs, some invariants accumulate bookkeeping information then require some final processing at transaction commit.

[2] We did not observe this problem because our fault injector corrupts metadata blocks but does not cause misdirected writes (see Section 5.2).

Datatype		#rules
A	Within superblock	23
B	Within block group descriptors (BGD)	5
C	Within a single inode	28
D	Within a single directory	14
E	Between inode and directory entries	5
F	Between inode and its block pointers	2
G	Between inode, inode bitmap, orphan list	3
H	Between block bitmap and block pointers	5
I	Between block, inode bitmap, BGD table	3
J	Between directories	4
K	Bad blocks inode	7
L	Extended attributes ACL	2

Table 2: Number of Ext3/SQCK rules by datatype

Datatype		#rules
A	Within tree block	2
B	Between parent and child tree blocks	3
C	Between extent tree and extents	3
D	Within an extent item in extent tree	2
E	Between inodes and file system trees	2
F	Between inode and directory entries	4
G	Between inodes, inode refs and dir. entries	2
H	Within directory entries	1
I	Between inode, data extents, checksum tree	6
J	Between inode and orphan items	1
K	Between root tree and file system trees	3
L	Between root tree and orphan items	1

Table 3: Number of Btrfs rules by datatype

3.1 Ext3 Immutable Fields, Range Checks

The ext3 fsck program checks for valid values in several fields of the superblock and group descriptor table (rows A and B in Table 2). Many of these fields are initialized when a file system is created and should never be modified by a running file system. Invariants on these fields are implemented by pattern matching a change record of the form [Superblock, _, immutable_field, _, _], where immutable_field is the name of the field that should not change, and _ matches any value. The existence of this record indicates that the field was modified, and signals a violation. Another similar class of consistency properties requires simple range checks on the values of given fields.

3.2 Ext3 Block Bitmap and Block Pointers

An important consistency properties in ext3 is that no data block may be doubly allocated, i.e., every block pointer (whether it is found in a live inode or indirect block) must be unique or 0. Checking this property would be expensive if we simply scanned all inodes and indirect blocks searching for another instance of the pointer.

The file system maintains this property without examining the entire disk state by using block allocation bitmaps (row H in Table 2), with the resulting consistency property being that "all live data blocks are marked in the block bitmap". The corresponding consistency invariant is that a transaction that makes a data block live (i.e., by adding a pointer to the block) must also contain a corresponding bit-flip (from 0 to 1) in the block bitmap within the same transaction, as shown below.

block pointer set to N from 0 ⇔ bit N set in bitmap (1)
block pointer set to 0 from N ⇔ bit N unset in bitmap (2)

These invariants involve relationships between different fields and require matching multiple change records. The left side of the first invariant is triggered by matching change records of the form [_, _ , block_pointer_field, 0, X], indicating a new pointer to block X. When such a match occurs, we insert a "new pointer" flag with key X into a rule-specific table. The right side of this (biconditional) invariant is triggered by matching [BBM, Y, _, 0, 1] records, indicating bit Y in the allocation bitmap is newly set. When this match occurs, we insert a "bit set" flag with key Y into the same table. During final processing, the implementation verifies that for each key in the table, both flags are set. Otherwise the invariant has been violated. For example, in the simple transaction shown in Figure 2, there is exactly one record matching each of the left and right sides of Invariant 1 shown above, and the values of X and Y are both 22717.

Invariants 1 and 2 ensure that when a block pointer is set, the corresponding bit in the bitmap is also set. However, we must also ensure that a pointer to the same block is set only once in a transaction, i.e., we must check for double allocation within a transaction. To do so, we simply count the number of times we see a block pointer set to a given block in the transaction:

block pointer set to N ⇒
(count(block pointer==N) in transaction)==1 (3)

3.3 Ext3 Directories

The inter-directory consistency properties essentially ensure that the directory tree forms a single, bidirected[3] tree (row J in Table 2). This complex consistency property requires two biconditional and two regular invariants. Whenever a directory is linked (or its ".." entry changes), Invariant 4 checks that the directory's parent (child) has the directory as its child (parent). This check also ensures that a directory does not have multiple parents. When a directory is unlinked (or moved), Invariant 5 checks that it is unlinked on both sides (although not shown, we also check that an unlinked directory is empty). When a directory's "." entry is updated, Invariant 6 checks that the "." entry points to itself.

[Dir, C, "..", _, P] ⇔ [Dir, P, nm, _, C] and (nm != "..") (4)

[3]A bidirected tree is the directed graph obtained from an undirected tree by replacing each edge by two directed edges in opposite directions.

[Dir, C, "..", P, _] ⇔ [Dir, P, nm, C, _] and (nm != "..") (5)
[Dir, D1, ".", _, D2] ⇒ D1==D2 (6)
[Dir, _, "..", _, P] ⇒ is_ancestor(ROOT, P) (7)

Finally, Invariant 7 checks that a directory update does not cause cycles. Invariants 4 and 5 do not prohibit cycles. For example, suppose that the file system allows the command "mv /a /a/b" to complete successfully. This update would be allowed by the Invariants 4 and 5, but it would create a disconnected cycle consisting of a and b. Invariant 7 checks for cycles when a directory's parent entry (the ".." entry) is updated. It ensures that the chain of parent directories eventually reaches the root directory, or a cycle is detected. The is_ancestor() primitive operates on the Recon metadata caches described in Section 4.

3.4 Btrfs Inode and Directory Entries

Metadata structures in btrfs are indexed by a 17-byte key consisting of the tuple (objectID, type, offset). ObjectID is roughly analogous to an inode number in ext3. The type field determines the type of the structure, and the meaning of "offset" depends on the type. Each key is unique within a btrfs tree, so the unique (type, id) pair for our change records consists of (type, (tree id, objectid, offset)).

A btrfs consistency property is that the inode associated with a directory item (that is, a btrfs directory entry) has a directory mode (row F in Table 3). An invariant derived from this property is that when we add a new directory item, there must exist an appropriate inode item after transaction commit. We can represent this as:

[DIR_ITEM, (T, I, _), _, ϕ, _] ⇒
  exists(T, I, INODE_ITEM, 0) and
  ISDIR(get_item(T, I, INODE_ITEM, 0).mode)

The left hand side matches a directory item within snapshot tree T and objectid I that is being newly created. This invariant asserts that 1) there is a matching inode item, and 2) its mode is of directory type. The exists() primitive returns true if the given item can be found in tree T, and the get_item primitive obtains the contents of the item, allowing us to check the mode. These primitives operate on the Recon metadata caches.

4 Implementation

We use the Linux device mapper framework to interpose on all file system I/O requests at the block layer, as shown in Figure 1. On a metadata block read, recon_read caches the block in the Recon read cache. This cache allows accessing the disk or the pre-update file-system metadata state efficiently. Its contents are trusted because its blocks have been verified previously. On a metadata block write, recon_write caches the updated block in the Recon write cache. The write cache may contain corrupt data and thus any code accessing this cache must perform careful validation. Both caches also store block-specific information such as the block-type map. Similar to a file system buffer cache, neither Recon cache persists across reboots.

4.1 Commit Process

At commit, our framework requires that 1) all transaction blocks must have been recorded using recon_write, and 2) recon_commit is called before the commit block reaches the disk. We can record blocks and detect commit either from the transaction subsystem (transaction-layer commit) or at the block layer (block-layer commit). With transaction-layer commit, the file system's transaction commit code is modified to invoke recon_write on the updated metadata blocks, and invoke recon_commit before writing the commit block. This method is simpler to implement, but it makes us dependent on the transaction layer code, such as JBD in ext3. In particular, it does not allow us to verify the ext3 checkpointing process.

With block-layer commit, recon_write could be invoked on all block writes. The challenge is to separate metadata blocks from data blocks because we do not want to cache every data block. However, we can only identify newly allocated metadata blocks at commit, making them hard to distinguish from data on each write. Fortunately, for ext3, metadata blocks are written to the journal, and thus we can ignore blocks that are not journaled. This approach requires interpreting journal writes at the block layer, which also helps detect commit. While this implementation is more complicated, it removes any dependency on the journaling code. For btrfs, metadata writes can be easily distinguished because they are directed to designated regions on disk called btrfs chunks. Btrfs commits occur when the superblock is written, which is easy to detect because the superblock is in a known location.

We have implemented both transaction-layer and block-layer commit, but currently we have only evaluated the transaction-layer commit implementation.

4.2 Cache Pinning and Eviction

We control the amount of memory used by the Recon caches with a simple LRU mechanism for replacing blocks from the read cache when it grows beyond a user-configurable limit. All read cache blocks are pinned during recon_commit processing to simplify implementation. We expect that recon_commit will run quickly because the blocks needed for commit processing have likely been read by the file system recently and so they will not need to be read from disk to populate the read cache. We pin the Recon write cache for the duration of the transaction because we will need these blocks for checking invariants. This approach is similar to the ext3 file system pinning its journal blocks for performance. However, we could unpin a block once it reaches disk, e.g., the journal in ext3.

After commit, the contents of the write cache are merged into the read cache, thus updating Recon's view

FS Recon API	Invoked on	
references	Read	provides type and id information for data structures in referenced blocks
process_write	Commit	provides type and id information for newly allocated metadata blocks
process_txn	Commit	generates change records
txn_check	Commit	checks invariants using change records and metadata read/write caches

Table 4: File-system specific Recon API

of file-system state, and the write cache is cleared. At this point, we can unpin the read cache because all the blocks in the cache are on disk (e.g., either in the journal or the checkpointed location in ext3). However, our transaction-layer commit implementation for ext3 does not track the location of blocks in the journal. To avoid evicting a block that may be in the journal, we keep a list of most recently updated blocks in the read cache. This list contains as many blocks as it takes to fill the journal and we pin these blocks. Once a block is evicted from this list, it must have been checkpointed, or else it would have been overwritten in the journal, and so we can unpin it.

4.3 File-System Specific Processing

Recon invokes file-system specific API functions for metadata interpretation and invariant checking, as shown in Table 4. The *references* function is invoked by recon_read to parse a metadata block and create block-type mappings for pointed-to blocks. This function is also used to distinguish between data and metadata on the read path.

The rest of the functions in Table 4 are invoked by recon_commit. The *process_write* function is similar to the references function but invoked on all the blocks in the write cache (i.e., each updated or newly allocated metadata block). This function must validate the updated blocks by checking that any pointers, strings and size fields within the block have reasonable values so that further processing is not compromised. Recon ignores unknown blocks and only processes updated blocks whose types are known. As unknown blocks become known, they are added to the queue of blocks being processed. At the end of write processing, if any unknown blocks remain, Recon signals a reachability invariant violation, as discussed in Section 2.3.

Once the block and data types within blocks are known, the *process_txn* function compares updated data structures with their previous versions to derive a set of change records. The previous version of a data structure is uniquely determined by the (type, id) pair of the change record. In ext3, the type is determined by block type and the id is typically an inode number or a block number. In btrfs, the type and id are determined by the tree and the key, as discussed in Section 3.4.

While the process of comparing data structures is clearly file-system specific, we found two common cases. When data structures have fixed size, such as inodes in ext3 and most items in btrfs, we use a simple byte-level diff that is driven by tables that describe the layout of the data structures. These tables are generated from the data structures using C macros. When data structures themselves contain sets of smaller items, such as directory entries in ext3, or extent items in btrfs, we use a set-intersection method to derive three sets consisting of new items, deleted items and modified items. Change records can be generated from these sets, using the identity of the containing item (e.g., directory inode) and some key as field name (such as the "name" for directory entries).

The *txn_check* function implements invariant checking as described in Section 3 with examples.

4.4 Handling Invariant Violation

The final problem for an online consistency checker like Recon is dealing with invariant violations. It is important to ensure that recovery from a violation is correct and so the safest strategy is to disable all further modifications to the file system to avoid corruption. The file system can then be unmounted and restarted manually or transparently to applications [26]. In this case, the file system is not corrupt but may have lost some data. If the ability to create a snapshot (e.g., a btrfs snapshot) is available, then a snapshot could be created immediately, the problem reported, and then we could continue running the file system. It is important to isolate the snapshot from the buggy file system, e.g., by directing all further writes to a separate partition. In this case, data is preserved but the file system may be corrupt. Finally, it may be possible to repair file system data structures dynamically [8].

5 Evaluation

In this section, we evaluate our Recon implementation for ext3 in terms of its 1) complexity, 2) ability to detect metadata corruption at runtime, and 3) its performance impact. Currently, we are finishing our btrfs implementation, and we plan to evaluate it in the near future.

5.1 Completeness and Complexity

We have implemented all of the checks performed by the e2fsck file system checker, as encapsulated by the SQCK rules, for the mandatory file system features. Overall, we need only 31 invariants (vs 101 SQCK rules) because some properties are easier to verify at runtime. For example, a large number of fields in the superblock and block group descriptors are protected with the simple invariant that they should not be changed by a running file system.

We also avoid explicit range check invariants in several cases because they are naturally embedded in other invariants that must check for setting or clearing of bits in bitmaps. There are a small number of properties on optional features that we do not check, such as OS-specific fields in inodes and the extended attributes ACLs.

Our entire system consists of 3.8k lines of C code (kLOC), as measured by the cloc [7] tool. Of these, 1.5 kLOC are in the generic framework which can be reused across file systems, 1.5 kLOC are for interpreting the ext3 metadata, and only 0.8 kLOC are involved in checking the invariants. Our dependence on the journal checkpointing code adds another 311 lines. The code required to do the checking is simpler than the file system code for several reasons. First, within the thread checking a transaction, we do not need to worry about concurrency, as the buffers we are examining are under the control of the journal. In contrast, the file system needs to be servicing multiple client threads. Second, the implementation of each invariant check is independent of the other checks because each rule uses its own data structures to keep track of properties that must be verified. Finally, the implementation of each rule is usually quite simple, requiring several lines of C to accumulate the necessary data and a few more (often just a single boolean expression) to verify.

5.2 Ability to Detect Corruption

Evaluating resiliency against metadata corruption is tricky. To best represent real-world corruption scenarios, we would either inject subtle bugs in the file-system or reproduce known bugs. However, subtle bugs (i.e., bugs not easily found in a heavily-used file system) are hard to design or reproduce. Reproducing known bugs is difficult as they often depend on specific kernel versions, combinations of loadable modules, concurrency levels, or workloads. Instead, we settled for deliberately injecting corruption of bytes within metadata blocks. This mimics the corruption that could result from several types of bugs (e.g., setting values in arbitrary fields incorrectly) both within the file system or in the overall kernel. We injected both type-specific corruption, where we target specific metadata block types and fields, and fully random corruption where we corrupt a sequence of 1 to 8 bytes within some number of blocks in a transaction.

Setup We compare Recon against e2fsck by corrupting metadata just before it is committed to the journal. We begin each corruption experiment by creating and populating a fresh file system, to ensure that there are no errors initially. Next, we start a process that creates a background of I/O operations (specifically we run a kernel compile and clean, repeatedly). The corruptor then sleeps for 20-90 seconds, wakes up, and performs the requested corruption (type-specific or random). We record

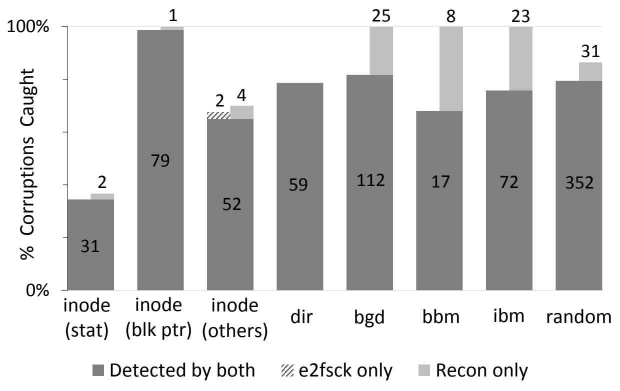


Figure 3: Comparison of corruption detection accuracy

the corruption performed and whether or not Recon detected it. Next, we allow the transaction to commit, and then immediately prevent any future writes. This step ensures that the corruption is limited to the bytes that we selected, rather than the result of the file system acting further on corrupt data. Next, we unmount the file system, run e2fsck on it, and record whether it found and repaired any errors. Finally, we run e2fsck a second time to see if the file system is clean after the repairs, and then reboot the system for the next experiment. For these experiments, we use a 4 GB file mounted as a loop device for our file system. This simplified the restoration of the file system following each corruption experiment.

Our corruption framework can only corrupt blocks that the file system is already modifying in some transaction. In particular, we never corrupt the superblock since the running file system never includes writes to it. We do not consider this to be a serious limitation to our test results since nearly all superblock corruptions would be trivially detected by Recon. Specifically, Recon protects most fields in the superblock with the invariant that they should not be modified at all, which is very easy to check.

Results Figure 3 summarizes the results of our corruption experiments. We show a wide bar and two stacked bars for each type of metadata corruption and random corruption. The wide bar shows the percent of corruptions (Y axis) that were caught by both e2fsck and Recon. The stacked bars show the percent of corruptions that were detected by only one checker. Numbers in the bars show the absolute number of corruptions detected.

For inodes, we present 3 sets of bars, representing different types of inode fields. The first group includes fields that are reported by “stat”, the second group consists of all the block pointer fields, and the third group consists of everything else. Our coverage is nearly identical to e2fsck in all cases. Many of the inode stat fields are unrelated to file system consistency (e.g. the timestamps and userids) and are permitted to change arbitrarily, making it hard to detect corruption with either checker. However, both checkers are effective at catching corruption of block pointers. Recon achieves 100% in this case because it checks all inodes in a block being written to disk while e2fsck ignores unused inodes. Although file system consistency is not affected by changes to unused inodes, it is still useful to detect this corruption because it indicates a bug in the system. For the final set of inode fields, e2fsck detects an invalid flag setting that Recon does not check in two runs, while Recon catches corruption of some unused inode flags and a corruption of the dir_acl field that appears valid when checked by e2fsck after the fact in four runs.

For directory entries (dir), both checkers detect the same corruptions, with neither checker detecting corruption of the name field. For the other metadata types, Recon is more effective than e2fsck at detecting corruption, largely because it is able to take other runtime behavior into account. For example, Recon achieves 100% detection for block group descriptor (bgd) corruption because most of these fields should not be changed by a running file system. Once corruption has reached the disk however, it is not always possible to distinguish the correct values from corrupted, but still valid, values. Similarly, Recon detects 100% of the block and inode bitmap (bbm and ibm, respectively) corruptions while e2fsck has a lower detection rate because it does not check unused parts of metadata blocks. For example, e2fsck does not check bits in the inode bitmap for non-existent inodes, or bits in the block bitmap for uninitialized block group descriptor table blocks. Recon's higher coverage on specific metadata fields leads to higher coverage for fully random corruption as well. We expect that adding the final set of ext3 invariants for OS-specific inode fields and extended attributes will help us detect all ext3 structural consistency violations. However, neither checker can achieve 100% accuracy because some of the corruptions hit fields unrelated to structural consistency.

After e2fsck performs repair, it still detects errors in 28 out of 731 cases (3.8%), when it is run a second time on the "repaired" file system. Two of these failures occurred after a single byte was corrupted in a single metadata block. In our experiments, we unmount the file system and check it with e2fsck immediately after the corrupted transaction is committed to the journal. In reality, it is likely that the file system would continue operation with bad data for some time, making the chances of successful repair even lower. In these cases, Recon's ability to prevent corruption from reaching the on-disk metadata is particularly valuable.

5.3 Performance

Setup All performance tests were done on a 1 TB ext3-formatted file system on a machine with 2GB total RAM and dual 3 GHz Xeon CPUs. We used the Linux port of FileBench (version 1.4.8.fsl.0.8) with the application emulation workload personalities[4]. We included the Networkfs personality, which supports a more sophisticated file system model, with a custom profile configured to match the metadata characteristics from a recent study of Windows desktops[17]. For Fileserver, we reduced the default file size to 32k to increase the metadata to data ratio in the file system. In all other cases, we used default parameter settings. Table 5 summarizes the basic characteristics of our benchmarks.[5] The metadata load varies widely across the benchmarks, spanning the range of Recon cache sizes, causing misses in the cache. In particular, the Fileserver benchmark uses over 25k directories. The metadata consumed by directory entry blocks alone is greater than 100MB. The inodes for the directories and files would consume approximately 70MB if they were stored compactly, but ext3 distributes allocation across different block groups, so unused inodes add to the metadata overhead. While the Networkfs benchmark involves more file data, the total number of files is lower because of the larger file size distribution.

Personality	**Settings**	**Data Size**
Webserver	nfiles=250k	3.9 GB
Webproxy	nfiles=500k	7.8 GB
Varmail	nfiles=250k	3.9 GB
Fileserver	nfiles=500k, filesize=32k	15.6 GB
MS-Networkfs	based on [17]	19.9 GB

Table 5: Benchmark Characteristics

The benchmarks are run for one hour for all workloads to ensure that we capture steady-state behavior with Recon. We report the performance of Recon compared to native ext3 for both the initial benchmark setup, which involves heavy metadata writes (Table 6), and the actual workload execution (Figure 4).

Our current transaction-layer commit implementation (described in Section 4) cannot evict blocks from our metadata cache that have not yet been checkpointed to the file system. Thus, the metadata cache size must be larger than the journal size. However, any memory consumed by Recon's metadata cache reduces the memory available for the file system cache by the same amount because Linux implements a shared page cache. We present results for three different cache/journal sizes, for both native and Recon performance. FileBench emulates workloads using a variety of random variables for file and operation selection. Thus, there is natural performance variation across runs. Since this is representative of behavior "in the wild", we report the average of 5 runs with error bars. All tests are done with cold caches on a freshly booted system.

[4]The OLTP personality did not work in the version we obtained.

[5]The full profile used in the experiments is available at http://csng.cs.toronto.edu/publications/260/get?file=/publication_files/210/recon-fast2012-workloads.tgz

	Cache=64MB, Journal=32MB			Cache=128MB, Journal=64MB			Cache=256MB, Journal=128MB		
Setup (seconds)	**Ext3**	**Recon**	**Ratio**	**Ext3**	**Recon**	**Ratio**	**Ext3**	**Recon**	**Ratio**
Webserver	2171.0±42.8	2903.2±45.7	133.7	1722.0±77.4	1668.6±36.7	96.9	1405.6±24.4	1340.2±29.6	95.3
Webproxy	229.4±26.0	323.0±24.3	140.8	212.8±13.5	243.4±23.5	114.4	227.2±19.5	224.4±24.0	98.8
Varmail	110.2±11.4	110.8±4.4	100.5	118.6±12.3	113.8±16.2	96.0	109.4±9.5	123.0±5.0	112.4
Fileserver	13728.5±694.2	17705.8±413.5	129.0	11487.2±849.8	12906.8±1316.8	112.4	9785.6±491.6	10374.8±928.8	106.0
Networkfs	2096.8±140.4	2113.8±119.2	100.8	1757.4±70.2	1893.0±73.0	107.7	1651.8±113.8	1719.4±31.5	104.1

Table 6: Setup time for benchmarks (lower is better)

Results During the benchmark setup, when many files are being created, there is a significant cost to Recon, particularly for small cache sizes. The dominating factor is I/O time for metadata cache misses because file creation quickly and repeatedly touches the entire working set of metadata. However, as the cache size increases, the impact is rapidly reduced. With a 128MB metadata cache, the added overhead of Recon is within the experimental error of ext3's native performance. The impact of Recon is less noticeable during normal benchmark operations. With our smallest metadata cache size (64MB), there is a worst case overhead of only 15% for Fileserver, which is generally reduced as the cache size increases. The one exception to this trend is the Networkfs personality (ms_nfs in Figure 4), where performance degrades with an increasing Recon cache size. We believe this is the result of memory pressure, as our increased metadata cache size decreases the amount of memory available to the file system buffer cache. Overall, a 128MB metadata cache with a 64MB journal gives the best results for all workloads, with only 8% degradation on average. In most cases, file system throughput with Recon is within the margin of error of ext3 performance. Given the growth in main memory sizes, these are quite modest memory requirements for the reliability benefits that Recon can deliver.

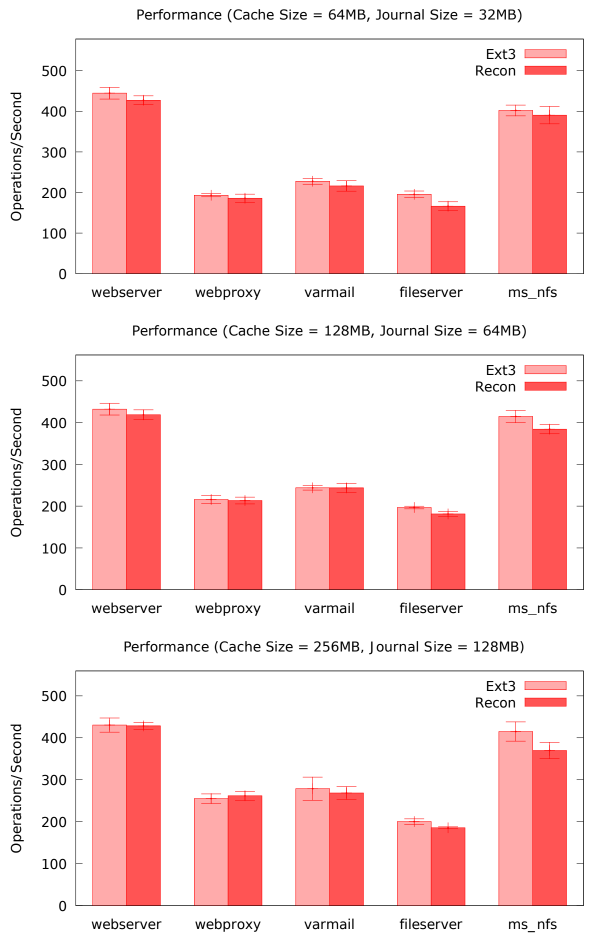


Figure 4: Performance on FileBench workloads for varying metadata cache sizes

6 Related Work

We discuss several areas of research that are closely related to this work, including methods for 1) handling file system bugs, 2) checking file system consistency, 3) interpreting file system semantics and verification.

6.1 Handling File System Bugs

File system bugs can be detected statically or at runtime. Bug finding tools, based on model checking [29, 31] and static analysis [21], have revealed scores of bugs in a variety of file systems. However, these tools cannot be relied upon to identify all bugs because they need to perform exhaustive evaluation. Furthermore, even when a bug is known, a bug fix may not be easily available, or easy to deploy in live systems [1]. These limitations can be addressed by tolerating bugs at runtime.

EnvyFS [3] applies N-version programming for detecting file system bugs. It uses the common VFS interface to pass each file system request received by the VFS layer to three child file systems. The results are then compared and the majority result is returned. EnvyFS avoids storing 3 data copies by using a customized single-instance store. Although EnvyFS is able to detect and in some cases repair errors introduced in child file systems, the run time overheads are significant because the operations must be issued to at least two file systems and the results compared before an answer is returned. Also, subtle differences in file system semantics can make it hard to compare results.

Membrane [26] proposes tolerating bugs by transparently restarting a failed file system. It assumes that file system bugs will lead to detectable, fail-stop crash failures. However, inconsistencies may have propagated to the on-disk metadata by the time the crash occurs. Our approach is complementary to Membrane, rather than waiting for the file system to crash, a restart could be initiated when Recon detects an inconsistent transaction.

6.2 Checking File System Consistency

SQCK [11] expresses the many complex checks performed by e2fsck as a set of compact SQL queries. It improves upon the repairs done by e2fsck by correcting the order in which repairs were performed and by using redundant file-system metadata ignored by e2fsck.

Chunkfs proposes reducing the consistency check time by breaking the file system into chunks that can be checked independently [13]. While this idea is appealing, unfortunately the chunks are not independent and thus cannot be checked truly independently. Specifically, pathnames can span chunks, and Chunkfs uses cross-chunk references to handle hard links and files that are larger than chunks or need allocation across chunks.

ZFS provides the ability to scrub disks and repair corrupt blocks that have redundant copies [4]. Scrubbing can detect latent hardware errors but does not necessarily detect software bugs, e.g., if the block has a consistency error but passes the checksum. NetApp filers can run some phases of the wafliron check program on an online system, but this process is resource intensive and time-taking.

6.3 File System Semantics and Verification

Semantically-smart disks use probing to gather detailed knowledge of file system behavior [24]. This knowledge is used at the block interface to transparently improve performance or enhance functionality, such as by implementing track-aligned extents and secure delete. This work builds on several ideas from semantically-smart disks.

The XN storage system of the Xok exokernel is designed to protect library file systems that manage their own disk blocks [15]. XN uses a file-system specific function called own(), similar to the Recon references() function, that returns the blocks controlled by a meta-data block. This function allows XN to verify that a file system can only access blocks that are allocated to it. XN can also use a file-system specific function called reboot() that traverses the entire file-system tree and detects whether the file system is crash consistent. This work shows that file-system consistency can be verified at runtime efficiently. File systems must use an extended block interface (e.g., allocate, read, write, deallocate) and provide block type information to XN and which allows easier verification, while Recon only requires the basic block interface (e.g., read, write) and infers file system information. Also, XN protects file systems from each other and may allow a file system to corrupt itself, while our focus is on protecting the file system from itself. Similar to XN, a type-safe disk extends the disk interface by exposing primitives for block allocation [23], which helps enforce invariants such as preventing accesses to unallocated blocks.

There has been significant work on discovering program invariants by capturing variable values at key points in a program to repair data structures [8] and to patch buggy deployed software [18]. We plan to apply these methods to learn file-system invariants and repair updates that cause invariant violations. Our work is influenced by runtime verification, a technique that applies formal analysis to the running system rather than its model [25, 5].

Our system can be viewed as a firewall with a set of rules that help protect disks from accesses that could compromise file-system integrity. Defining and implementing these rules in a high-level language, such as the Linux iptables rules [22], is an avenue for future work.

7 Conclusions and Future Work

The Recon system protects file system metadata from buggy file system operations. It uses two key ideas, using *commit points* to verify *consistency invariants*. Modern file systems aim to ensure file system consistency at commit points. Consistency invariants are declarative statements that must be satisfied at these points before data is committed or else the file system may get corrupted. We reuse the consistency rules used by a file system checker to derive the invariants. As a result, Recon detects random corruption at runtime as effectively as the file system checker. It has low overhead because the data it interprets has likely been recently accessed by the file system.

A system that checks the file system is easier to implement correctly than the file system itself. When checking a transaction, we do not need to worry about concurrency because the buffers we are examining are under our control. In contrast, the file system needs to be servicing multiple client threads. Also, each invariant is independent because it uses its own data structures to keep track of the properties that must be checked, and we find that the implementation of each rule usually quite simple. The bulk of the complexity lies in interpreting metadata structures. We plan to develop a systematic way to describe and interpret these structures.

While an offline checker can only make decisions based on the current file system state, Recon can also observe the file system operations in progress. We plan to investigate whether this allows detecting certain operational bugs unrelated to file system consistency, e.g., updates to userid or timestamp fields.

8 Acknowledgments

We thank the anonymous reviewers and our shepherd, Junfeng Yang, for many insightful comments. This research was supported by NSERC through the Discovery Grants program.

References

[1] ARNOLD, J., AND KAASHOEK, M. F. Ksplice: automatic rebootless kernel updates. In *Proceedings of the ACM SIGOPS European Conference on Computer Systems (Eurosys)* (2009), pp. 187–198.

[2] BAIRAVASUNDARAM, L. N., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., GOODSON, G. R., AND SCHROEDER, B. An analysis of data corruption in the storage stack. *Transactions of Storage 4*, 3 (2008), 1–28.

[3] BAIRAVASUNDARAM, L. N., SUNDARARAMAN, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Tolerating file-system mistakes with envyfs. In *Proceedings of the USENIX Technical Conference* (June 2009).

[4] BONWICK, J., AND MOORE, B. ZFS - The Last Word in File Systems. http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf.

[5] CHEN, F., AND ROŞU, G. Mop: an efficient and generic runtime verification framework. In *Proceedings of the ACM OOPSLA* (2007), pp. 569–588.

[6] CUSTER, H. *Inside the Windows NT File System*. Microsoft Press, 1994.

[7] DANIAL, A. CLOC – Count Lines of Code. http://cloc.sourceforge.net/.

[8] DEMSKY, B., AND RINARD, M. C. Goal-directed reasoning for specification-based data structure repair. *IEEE Transactions on Software Engineering 32*, 12 (2006), 931–951.

[9] GANGER, G. R., MCKUSICK, M. K., SOULES, C. A. N., AND PATT, Y. N. Soft updates: a solution to the metadata update problem in file systems. *ACM Transactions on Computer Systems 18*, 2 (2000), 127–153.

[10] GUNAWI, H. S., PRABHAKARAN, V., KRISHNAN, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Improving file system reliability with I/O shepherding. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)* (2007), pp. 293–306.

[11] GUNAWI, H. S., RAJIMWALE, A., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. SQCK: A declarative file system checker. In *Proceedings of the Operating Systems Design and Implementation (OSDI)* (Dec. 2008).

[12] HAGMANN, R. Reimplementing the Cedar file system using logging and group commit. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)* (Nov. 1987).

[13] HENSON, V., VAN DE VEN, A., GUD, A., AND BROWN, Z. Chunkfs: Using divide-and-conquer to improve file system reliability and repair. In *Proceedings of the Workshop on Hot Topics in System Dependability (HotDep)* (2006).

[14] HITZ, D., LAU, J., AND MALCOLM, M. File system design for an NFS file server appliance. In *Proceedings of the USENIX Technical Conference* (1994).

[15] KAASHOEK, F. M., ENGLER, D. R., GANGER, G. R., BRICENO, H. M., HUNT, R., MAZIKRES, D., PINCKNEY, T., GRIMM, R., JANNOTTI, J., , AND MACKENZIE, K. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)* (1997), pp. 52–65.

[16] MASON, C., AND ET AL. Btrfs. http://btrfs.wiki.kernel.org.

[17] MEYER, D. T., AND BOLOSKY, W. J. A study of practical deduplication. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2010).

[18] PERKINS, J. H., KIM, S., LARSEN, S., AMARASINGHE, S. P., BACHRACH, J., CARBIN, M., PACHECO, C., SHERWOOD, F., SIDIROGLOU, S., SULLIVAN, G., WONG, W.-F., ZIBIN, Y., ERNST, M. D., AND RINARD, M. C. Automatically patching errors in deployed software. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)* (2009), pp. 87–102.

[19] PRABHAKARAN, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Model-based failure analysis of journaling file systems. In *Proceedings of the IEEE Dependable Systems and Networks (DSN)* (2005), pp. 802–811.

[20] PRABHAKARAN, V., BAIRAVASUNDARAM, L. N., AGRAWAL, N., GUNAWI, H. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Iron file systems. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)* (2005), pp. 206–220.

[21] RUBIO-GONZÁLEZ, CINDY, GUNAWI, S., H., LIBLIT, B., ARPACI-DUSSEAU, H., R., ARPACI-DUSSEAU, AND C., A. Error propagation analysis for file systems. In *Proceedings of the ACM SIGPLAN conference on programming language design and implementation (PLDI)* (2009), pp. 270–280.

[22] RUSSELL, R. Iptables. http://en.wikipedia.org/wiki/Iptables.

[23] SIVATHANU, G., SUNDARARAMAN, S., AND ZADOK, E. Type-safe disks. In *Proceedings of the Operating Systems Design and Implementation (OSDI)* (2006), pp. 15–28.

[24] SIVATHANU, M., PRABHAKARAN, V., POPOVICI, F. I., DENEHY, T. E., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Semantically-smart disk systems. In *USENIX Conference on File and Storage Technologies (FAST)* (2003), pp. 73–88.

[25] SOKOLSKY, O., SAMMAPUN, U., LEE, I., AND KIM, J. Run-time checking of dynamic properties. *Electronic Notes in Theoretical Computer Science 144* (May 2006), 91–108.

[26] SUNDARARAMAN, S., SUBRAMANIAN, S., RAJIMWALE, A., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND SWIFT, M. M. Membrane: Operating system support for restartable file systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2010).

[27] SWEENEY, A., DOUCETTE, D., HU, W., ANDERSON, C., NISHIMOTO, M., AND PECK, G. Scalability in the XFS file system. In *Proceedings of the USENIX Technical Conference* (1996), pp. 1–14.

[28] TWEEDIE, S. C. Journalling the ext2fs filesystem. In *Proceedings of the 4th Annual Linux Expo* (May 1998).

[29] YANG, J., SAR, C., AND ENGLER, D. Explode: a lightweight, general system for finding serious storage system errors. In *Proceedings of the Operating Systems Design and Implementation (OSDI)* (2006).

[30] YANG, J., SAR, C., TWOHEY, P., CADAR, C., AND ENGLER, D. Automatically generating malicious disks using symbolic execution. In *Proceedings of the IEEE Symposium on Security and Privacy* (2006), pp. 243–257.

[31] YANG, J., TWOHEY, P., ENGLER, D., AND MUSUVATHI, M. Using model checking to find serious file system errors. *ACM Transactions on Computer Systems 24*, 4 (2006), 393–423.

[32] 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 *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2010).

Understanding Performance Implications of Nested File Systems in a Virtualized Environment

Duy Le[1], Hai Huang[2], and Haining Wang[1]

[1]*The College of William and Mary, Williamsburg, VA 23185, USA*
[2]*IBM T. J. Watson Research Center, Hawthorne, NY 10532 USA*

Abstract

Virtualization allows computing resources to be utilized much more efficiently than those in traditional systems, and it is a strong driving force behind commoditizing computing infrastructure for providing cloud services. Unfortunately, the multiple layers of abstraction that virtualization introduces also complicate the proper understanding, accurate measurement, and effective management of such an environment. In this paper, we focus on one particular layer: storage virtualization, which enables a host system to map a guest VM's file system to almost any storage media. A flat file in the host file system is commonly used for this purpose. However, as we will show, when one file system (guest) runs on top of another file system (host), their nested interactions can have unexpected and significant performance implications (as much as 67% degradation). From performing experiments on 42 different combinations of guest and host file systems, we give advice on how to and how not to nest file systems.

1 Introduction

Virtualization has significantly improved hardware utilization, thus, allowing IT services providers to offer a wide range of application, platform and infrastructure solutions through low-cost, commoditized hardware (e.g., Cloud [1, 5, 11]). However, virtualization is a double-edged sword. Along with many benefits it brings, virtualized systems are also more complex, and thus, more difficult to understand, measure, and manage. This is often caused by layers of abstraction that virtualization introduces. One particular type of abstraction, which we use often in our virtualized environment but have not yet fully understood, is the nesting of file systems in the guest and host systems.

In a typical virtualized environment, a host maps regular files as virtual block devices to virtual machines (VMs). Completely unaware of this, a VM would format the block device with a file system that it thinks is the most suitable for its particular workload. Now, we have two file systems – a host file system and a guest file system – both of which are completely unaware of the existence of the other layer. Figure 1 illustrates such a scenario. The fact that there is one file system below another complicates an already delicate situation, where file systems make certain assumptions, based on which, optimizations are made. When some of these assumptions are no longer true, these optimizations will no longer improve performance, and sometimes, will even hurt performance. For example, in the guest file system, optimizations such as placing frequently used files on outer disk cylinders for higher I/O throughput (e.g., NTFS), de-fragmenting files (e.g., QCoW [7]), and ensuring meta-data and data locality, can cause some unexpected effects when the real block allocation and placement decisions are done at a lower level (i.e., in the host).

An alternative to using files as virtual block devices is to give VMs direct access to physical disks or logical volumes. However, there are several benefits in mapping virtual block devices as files in host systems. First, using files allows storage space overcommit when they are thinly provisioned. Second, snapshotting a VM image using copy-on-write (e.g., using QCoW) is simpler at the file level than at the block level. Third, managing and maintaining VM images and snapshots as files is

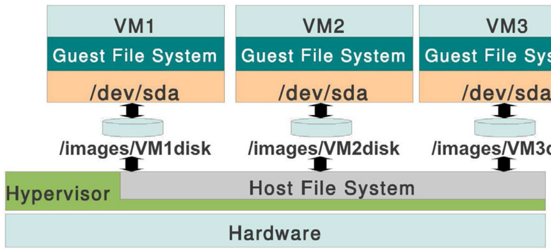


Figure 1: Scenario of nesting of file systems.

also easier and more intuitive as we can leverage many existing file-based storage management tools. Moreover, the use of nested virtualization [6, 15], where VMs can act as hypervisors to create their own VMs, has recently been demonstrated to be practical in multiple types of hypervisors. As this technique encourages more layers of file systems stacking on top of one another, it would be even more important to better understand the interactions across layers and their performance implications.

In most cases, a file system is chosen over other file systems primarily based on the expected workload. However, we believe, in a virtualized environment, the guest file system should be chosen based on not only the workload but also the underlying host file system. To validate this, we conduct an extensive set of experiments using various combinations of guest and host file systems including Ext2, Ext3, Ext4, ReiserFS, XFS, and JFS. It is well understood that file systems have different performance characteristics under different workloads. Therefore, instead of comparing different file systems, we compare the same guest file system among different host file systems, and vice versa. From our experiments, we observe significant I/O performance differences. An improper combination of guest and host file systems can be disastrous to performance; but with an appropriate combination, the overhead can be negligible.

The main contributions of this paper are summarized as follows.

- A quantitative study of the interactions between guest and host file systems. We demonstrate that the virtualization abstraction at the file system level can be more detrimental to the I/O performance than it is generally believed.
- A detailed block-level analysis of different combinations of guest/host file systems. We uncover the reasons behind I/O performance variations in different file system combinations and suggest various tuning techniques to enable more efficient interactions between guest and host file systems to achieve better I/O performance.

From our experiments, we have made the following interesting observations: (1) for write-dominated workloads, journaling in the host file system could cause significant performance degradations, (2) for read-dominated workloads, nested file systems could even improve performance, and (3) nested file systems are not suitable for workloads that are sensitive to I/O latency. We believe that more work is needed to study performance implications of file systems in virtualized environments. Our work takes a first step in this direction, and we hope that these findings can help file system designers to build more adaptive file systems for virtualized environments.

The remainder of the paper is structured as follows. Section 2 surveys related works. Section 3 presents macro-benchmarks to understand the performance implications of nesting file systems under different types of workloads. Section 4 uses micro-benchmarks to dissect the interactions between guest and host file systems and their performance implications. Section 5 discusses significant consequences of nested file systems with proposed techniques to improve I/O performance. Finally, Section 6 concludes the paper.

2 Related Work

Virtualizing I/O, especially storage, has been proven to be much more difficult than virtualizing CPU and memory. Achieving bare-metal performance from virtualized storage devices has been the goal of many past works. One approach is to use para-virtualized I/O device drivers [26], in which, a guest OS is aware of running inside of a virtualized environment, and thus, uses a special device driver that explicitly cooperates with the hypervisor to improve I/O performance. Examples include KVM's VirtIO driver [26], Xen's para-virtualized driver [13], and VMware's guest tools [9]. Additionally, Jujjuri *et al.* [22] proposed to move the para-virtualization interface up the stack to the file system level.

The use of para-virtualized I/O device drivers is almost a de-facto standard to achieve any reasonable I/O performance, however, Yassour *et al.* [32] explored an alternative solution that gives guest direct access to physical devices to achieve near-native hardware performance. In this paper, we instead focus on the scenario where virtual disks are mapped to files rather than physical disks or volumes. As we will show, when configured correctly, the additional layers of abstraction introduce only limited overhead. On the other hand, having these abstractions can greatly ease the management of VM images.

Similar to nesting of file systems, I/O schedulers are also often used in a nested fashion, which can result in suboptimal I/O scheduling decisions. Boutcher and Chandra [17] explored different combinations of I/O schedulers in guest and host systems. They demonstrated that the worst case combination provides only 40% throughput of the best case. In our experiments, we use the best combination of I/O schedulers found in their paper but try different file system combinations, with the focus on performance variations caused only by file system artifacts. Whereas, for performance purposes, there is no benefit to performing additional I/O scheduling in the host, it has a significant impact on inter-application I/O isolation and fairness as shown in [23]. Many other works [18, 19, 25, 27] have also studied the impact of nested I/O schedulers on performance, fairness, and iso-

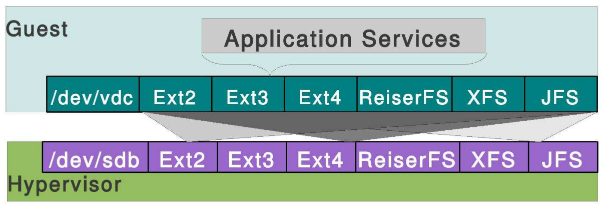


Figure 2: Setup for macro-level experimentation

lation, and these are orthogonal to our work in the file system space.

When a virtual disk is mapped to an image file, the data layout of the image file can significantly affect its performance. QCOW2 [7], VirtualBox VDI [8], and VMware VMDK [10] are some popular image formats. However, as Tang [31] pointed out, these formats unnecessarily mix the function of storage space allocation with the function of tracking dirty blocks. Tang presented an FVD image format to address this issue and demonstrated significant performance improvements for certain workloads. Various techniques [16, 20, 30] to dynamically change the data layout of image files, depending on the usage patterns, have also been proposed. Suzuki *et al.* [30] demonstrated that by co-locating data blocked used at boot time, a virtual machine can boot much faster. Bhadkamkar *et al.* [16] and Huang *et al.* [20] exploited data replication techniques to decrease the distance between temporally related data blocks to improve I/O performance. Sivathanu *et al.* [29] studied the performance effect of the image file placed at different locations of a disk.

I/O performance in storage virtualization can be impacted by many factors, such as device driver, I/O scheduler, and image format. To the best of our knowledge, this is the first work that studies the impact of the choice of file systems in guest and host systems in a virtualization environment.

3 Macro-benchmark Results

To better understand the performance implications caused by guest / host file system interactions, we take a systematic approach in our experimental evaluation. First, we exercise macro-benchmarks to understand the potential performance impact of nested file systems on realistic workloads, from which, we were able to observe significant performance impact. In Section 4, we use micro-benchmarks coupled with low-level I/O tracing mechanisms to investigate the underlying cause.

3.1 Experimental Setup

As there is no single "most common" or "best" file system to use in the hypervisor or guest VMs, we conduct

	Hardware	**Software**
Host	Pentium D 3.4GHz, 2GB RAM	Ubuntu 10.04 (2.6.32-33)
	80GB WD 7200 RPM SATA (**sda**)	qemu-kvm 0.12.3
	1TB WD 7200 RPM SATA (**sdb**)	libvirt 0.9.0
Guest	Qemu 0.9, 512MB RAM	Ubuntu 10.04 (2.6.32-33)

Table 1: Testbed setup

our experiments using all possible combinations of popular file systems on Linux (i.e., Ext2, Ext3, Ext4, ReiserFS, XFS, and JFS) in both the hypervisor and guest VMs, as shown in Figure 2. A single x86 64-bit machine is used to run KVM [24] at the hypervisor level, and QEMU [14] is used to run guest VMs [1]. To reflect typical enterprise setting, each guest VM is allocated a single dedicated processor core. More hardware and software configuration settings are listed in Table 1.

The entire host OS is installed on a single disk (`sda`) while another single disk (`sdb`) is used for experiments. We create multiple equal-sized partitions from `sdb`, each corresponding to a different host file system. Each partition is then formatted using the default parameters of the host file system's `mkfs*` command and is mounted using the default parameters of `mount`. In the newly created host file system, we create a flat file and expose this flat file as the logical block device to the guest VM, which in turn, further partitions the block device, having each corresponding to a different guest file system. By default, `virtio` [26] is used as the block device driver for the guest VM and we consider `write-through` as a caching mode for all backend storages. The end result is the guest VM having access to all combinations of guest and host file systems. Table 2 shows an example of our setup: a file created on `/dev/sdb3`, which is formatted as Ext3, is exposed as a logical block device `vdc` to the guest VM, which further partitions `vdc` into `vdc2`, `vdc3`, `vdc4`, etc. for different guest file systems. Note that all disk partitions of the hypervisor (`sdb*`) and the guest (`vdc*`) are properly aligned using `fdisk` to avoid most of the block layer interference caused by misalignment problems.

In addition to the six host file systems, we also create a raw disk partition that is directly exposed to the guest VM and is labeled as *Block Device (BD)* in Table 2. This allows a guest file system to sit directly on top of a physical disk partition without the extra host file system layer. This special case is used as our baseline to demonstrate how large (or how small) of an overhead the host file system layer induces. However, there are some side effects to this particular setup, and namely, the file systems being created on outer disk cylinders will have higher I/O throughput than those created on inner cylinders. For-

[1] Similar performance variations are observed in the experiments with other hypervisors including Xen and VMWare, which are shown in Appendix.

Host file system			
Devices	**#Blocks (x10^6)**	**Speed(MB/s)**	**Type**
sdb2	60.00	127.64	Ext2
sdb3	60.00	127.71	Ext3
sdb4	60.00	126.16	Ext4
sdb5	60.00	125.86	ReiserFS
sdb6	60.00	123.47	XFS
sdb7	60.00	122.23	JFS
sdb8	60.00	121.35	Block Device

Guest file system		
Device	**#Blocks x10^6**	**Type**
vdc2	9.27	Ext2
vdc3	9.26	Ext3
vdc4	9.27	Ext4
vdc5	9.28	ReiserFS
vdc6	9.27	XFS
vdc7	9.08	JFS

Table 2: Physical and logical disk partitions

Services	# Files	# Threads	File size	I/O size
File server	50,000	50	128KB	16KB-1MB
Web server	50,000	100	16KB	512KB
Mail server	50,000	16	8-16KB	16KB
DB server	8	200	1GB	2KB

Table 3: Parameters for Filebench workloads

tunately, as each disk partition created at the hypervisor level is 60GB, only a portion of the entire disk is utilized and thus limits this effect. Table 2 also shows the results of running `hdparm` on each disk partition. The largest throughput difference between any two partitions is only about 5%, which is fairly negligible.

The choice of I/O scheduler at host and guest levels can significantly impact performance [17, 21, 27, 28]. As file system is the primary focus of this paper, we used `CFQ` scheduler in the host and `Deadline` scheduler in the guest as these schedulers were shown to be the top performers in their respective domains by Boutcher and Chandra [17].

3.2 Benchmarks

We use Filebench [3] to generate macro-benchmarks of different I/O transaction characteristics controlled by predefined parameters, such as the number of files to be used, average file size, and I/O buffer size. Since Filebench supports a synchronization between threads to simulate concurrent and sequential I/Os, we use this tool to create four server workloads: a file server, a web server, a mail server, and a database server. The specific parameters of each workload are listed in Table 3, showing that the experimental working set size is configured to be much larger than the size of the page cache in the VM. The detailed description of these workloads is as follows.

- **File server:** Emulates a NFS file service. File operations are a mixture of `create`, `delete`, `append`, `read`, `write`, and `attribute` on files of various sizes.

- **Web server:** Emulates a web service. File operations are dominated by reads: `open`, `read`, and `close`. Writing to the web log file is emulated by having one `append` operation per `open`.

- **Mail server:** Emulates an e-mail service. File operations are within a single directory consisting of I/O sequences such as `open/read/close`, `open/append/close`, and `delete`.

- **Database server:** Emulates the I/O characteristic of Oracle 9i. File operations are mostly `read` and `write` on small files. To simulate database logging, a stream of synchronous `writes` is used.

3.3 Macro-benchmark Results

Our main objective is to understand how much of a performance impact nested file systems have on different types of workloads, and whether or not the impact can be lessened or avoided. As mentioned before, we use all combinations of six popular file systems in both the hypervisor and guest VMs. For comparison purpose, we also include one additional combination, in which the hypervisor exposes a physical partition to guest VMs as a virtual block device. This results in 42 (6×7) different combinations of storage / file system configurations.

The performance results are shown in Figures 3 and 6, in terms of I/O throughput and I/O latency, respectively. Each sub-figure consists of a left and a right side. The left side shows the performance results when the guest file systems are provisioned directly on top of raw disk partitions in the hypervisor. These are expressed in absolute numbers (i.e., MB per second for throughput or millisecond for latency) and are used as our baseline. The right side shows the relative performance (to the baseline numbers) of the guest file systems when they are provisioned as files in the host file system. In these figures, each column group represents a different storage option

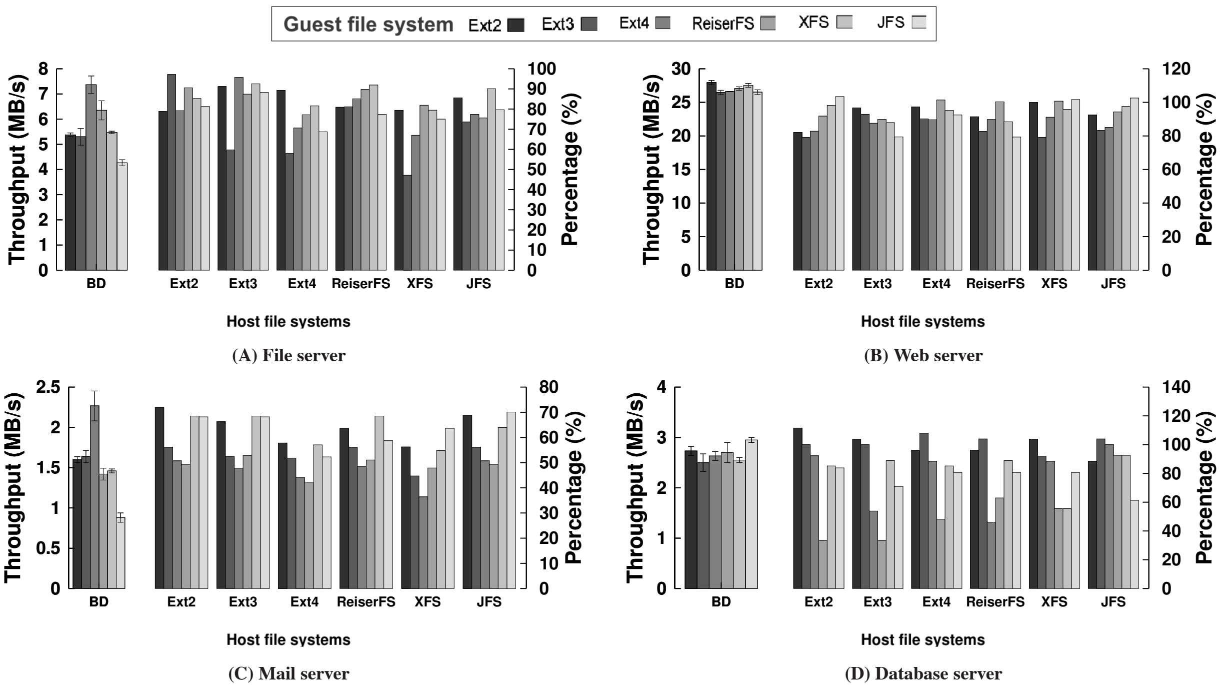


Figure 3: I/O throughput for Filebench workloads (**higher is better**)

in the hypervisor, and each column within the group represents a different storage option in the guest VM.

3.3.1 Throughput

The baseline numbers (leftmost column group) show the intrinsic characteristics of various file systems under different types of workloads. These characteristics indicate that some file systems are more efficient on large files than small files, while some file systems are more efficient at reading than writing. As an example, when ReiserFS runs on top of BD, its throughput under the web server workload (27.2 MB/s) is much higher than that under the mail server workload (1.4MB/s). These properties of file systems are well understood, and how one would choose which file system to use is a straightforward function of the expected I/O workload. However, in a virtualized environment where nested file systems are often used, the decision becomes more difficult. Based on the experimental results, we make the following observations:

(1) A guest file system's performance varies significantly under different host file systems. Figure 3(B) shows an example of the database workload. When ReiserFS runs on top of Ext2, its throughput is reduced by 67% compared to its baseline number. However, when it runs on top of JFS, its I/O performance is not impacted at all. We use coefficient of variance to quantify how differently a guest file system' performance is affected by different host file systems, which is shown in Figure 4. For

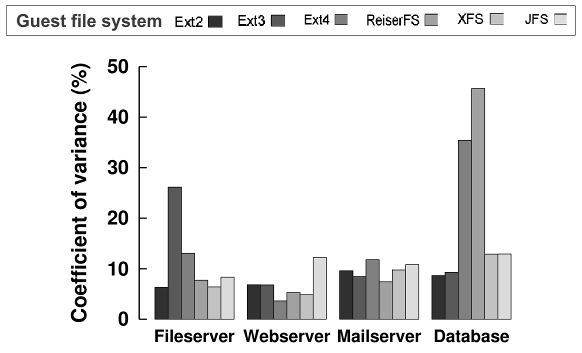


Figure 4: Coefficient of variance of guest file systems' throughput under Filebench workloads across different host file systems.

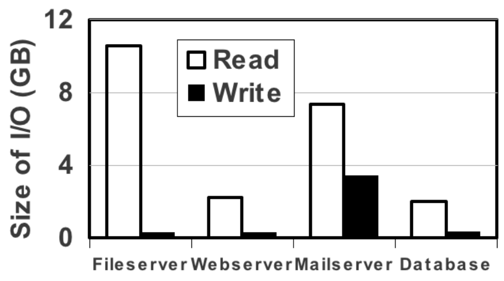


Figure 5: Total I/O transaction size of Filebench workloads

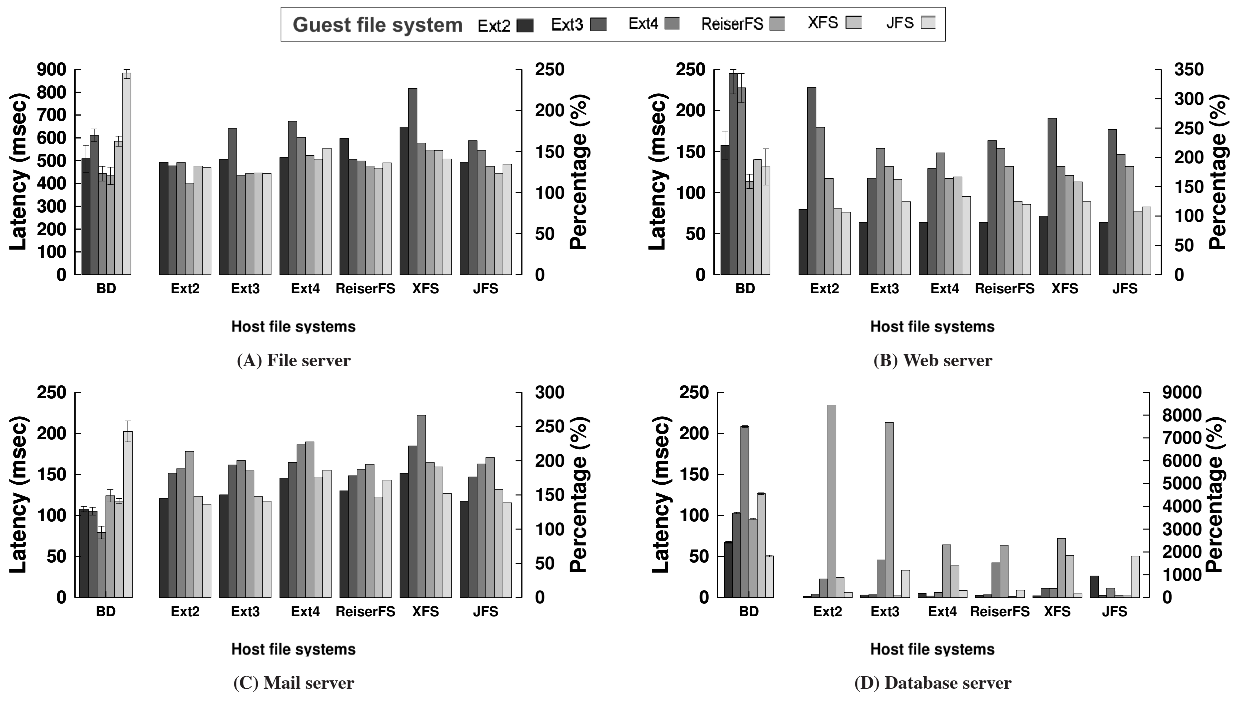


Figure 6: **I/O latency** of guest file systems under different workloads (**lower is better**)

each workload, a variance number is calculated based on relative performance values of a guest file system when it runs on top of different host file systems. Our results show that the throughput of ReiserFS experiences a large variation (45%) under the database workload, while that of Ext4 varies insignificantly (4%) under the web server workload. The large variance numbers indicate that having the right guest/host file system combination is critical to performance, and having a wrong combination can result in serious performance degradation. For instance, under the database workload, ReiserFS/Ext2 is a right combination, but ReiserFS/JFS is a wrong combination.

(2) A host file system impacts different guest file systems' performance differently. Similar to the previous observation, a host file system can have a different impact on different guest file systems' performance. Figure 3(A) shows an example of the file server workload. When Ext2 runs on top of Ext3, its throughput is slightly degraded by about 10%. However, when Ext3 runs on top of Ext3, the throughput is reduced by 40%. Based on results of coefficient of variance of guest file systems' throughputs shown in Figure 4, we observe that this bi-directional dependency between guest and host file systems again stresses the importance of choosing the right guest/host file system combination.

(3) A right guest file system/host file system combination can produce minimal performance degradation. Also based on results shown in Figure 4, one can also observe how badly performance can be impacted when a wrong combination of guest/host file system is chosen. However, it is possible to find a guest file system whose performance loss is the lowest. For example, the results of the mail server workload show that once Ext2 runs on top of Ext2, its throughput degradation is the lowest (by 46%).

(4) The performance of nested file systems is affected much more by write than read operations. As one can see in Figure 3, *all* the combinations of nested file systems perform poorly for the mail server workload, unlike the other three workloads. We study the detailed disk traces from these workloads by examining request queuing time, request merging, request size, etc., and find that the mail server workload is only significantly different from the others in having a much higher proportion of writes than reads, as shown in Figure 5. We will use micro-benchmarks in Section 4 to describe the reasons behind this behavior.

3.3.2 Latency

The latency results are illustrated in Figure 6. Similar to I/O throughput, latency is also deteriorated when guest file systems are provisioned on top of host file systems rather than raw partitions. Whereas the impact to throughput can be minimized (for some workloads) by choosing the right combinations of guest/host file system, latency is much more sensitive to nesting of file systems. In comparison to the baseline, the latency of each guest file system varies in a certain range when it

Description	Parameters
Total I/O size	5 GB
I/O parallelism	255
Block size	8 KB
I/O pattern	Random/Sequential
I/O mode	Native asynchronous I/O

Table 4: FIO benchmark parameters

runs on top of different host file systems. Even for the lowest cases, latency is increased by 5-15% across the board (e.g., Ext2 guest file system under the web server workload). Coefficient of variance for latency is similar to that of throughput shown in Figure 4. However, for latency sensitive workloads, like the database workload, such a significant increase in I/O response time could be unacceptable.

4 Micro-benchmarks Results

We first study nested file systems using a micro-level benchmark *FIO* [4]. Based on the experimental results, we further conduct an analysis at the block layer on the guest VM and the hypervisor, respectively, using an I/O tracing mechanism [2].

4.1 Benchmark

We use FIO as a micro-level benchmark to examine disk I/O workloads. As a highly configurable benchmark, FIO defines a test case based on different I/O transaction characteristics, such as total I/O size, block size, number of I/O parallelism, and I/O mode. Here our focus is on the performance variation of primitive I/O operations, such as *read* and *write*. With the combination of these I/O operations and two I/O pattens, *random* and *sequential*, we design four test cases: random read, random write, sequential read, and sequential write. The specific I/O characteristics of these test cases are listed in Table 4.

4.2 Experimental Results

On the same testbed, the experiments are conducted with many small files, which create a 5GB of total data footprint for each workload. Figures 7 and 8 show the performance in both sequential and random I/Os. Based on the experimental results, we make two observations:

- **The performance of those workloads that are dominated by read operations is largely unaffected by nested file systems.** The performance impact is weakly dependent on guest/host file systems. More interestingly, for sequential reads, in a few scenarios, a nested file system can even improve I/O performance (e.g., by 34% for Ext3/JFS).
- **The performance of those workloads that are dominated by write operations is heavily affected by nested file systems.** The performance impact varies in both random and sequential writes, with higher variations in sequential writes. In particular, a host file system like XFS can degrade the performance by 40% for both random and sequential writes. As a result, it is important to understand the root cause of this performance impact, especially on the sequential write dominated workload.

To interpret these observations, our analysis will focus on sequential workloads and the performance implication across certain guest/host file system combinations. For this set of experiments with micro-benchmark, due to space constraints, we only concentrate on deciphering the I/O behavior of these representative file system combinations. Although only a few combinations are considered, principles used here are applicable to other combinations as well.

For sequential read workloads, we attempt to uncover the reasons behind the significant performance improvement on the *right* guest/host file system combinations. We select the combinations of **Ext3/JFS** and **Ext3/BD** for analysis. For sequential write workloads, we try to understand the root cause of the significant performance variations in the scenarios of (1) different guest file systems running on the same host file system and (2) the same guest file system operating on different host file systems. We analyze three guest file system/host file system combinations: **Ext3/ReiserFS**, **JFS/ReiserFS**, and **JFS/XFS**. Here Ext3/ReiserFS and JFS/ReiserFS are used to examine how different guest file systems can affect performance differently on the same host file system, while JFS/ReiserFS and JFS/XFS are used to examine how different host file systems can affect performance differently on the same guest file system.

4.3 I/O Analysis

To understand the underlying cause of the performance impact due to nesting of file systems, we use blktrace to record I/O activities at both the guest and hypervisor levels. The resulting trace files are stored on another device, thus increasing only 3-4% CPU utilization. Therefore, the interference with our benchmarks from such an I/O recoding is negligible. Blktrace keeps detailed account of each I/O request from start to finish as it goes through various I/O states (e.g., put the request onto an I/O queue, merge with an existing request, and wait on the I/O queue). The I/O states that are of interest to us in this study are described as follows.

- Q: a new I/O request is *queued* by an application.

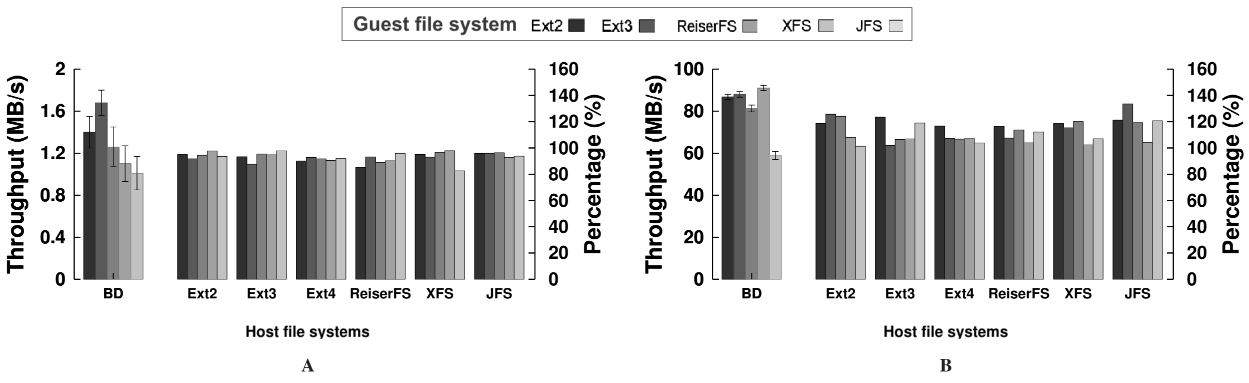


Figure 7: **I/O throughput** of guest file systems in **reading** files. (A): random and (B) sequential

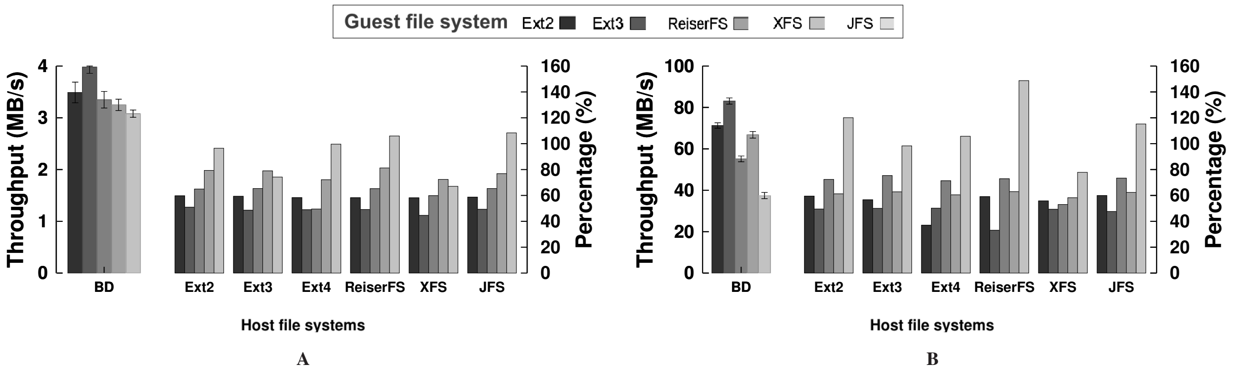


Figure 8: **I/O throughput** of guest file systems in **writing** files. (A): random and (B) sequential

- I: the I/O request is *inserted* into an I/O scheduler queue.
- D: the I/O request is being served by the *device*.
- C: the I/O request has *completed* by the device.

Blktrace records the timestamp when an I/O request enters a new state, so it is trivial to calculate the amount of time the request spends in each state (i.e., Q2I, I2D, and D2C). Here Q2I is the time it takes to insert/merge a request onto a request queue. I2D is the time it takes to idle on the request queue waiting for merging opportunities. D2C is the time it takes for the device to serve the request. The sum of Q2I, I2D, and D2C is the total processing time of an I/O request, which we denote as Q2C.

4.3.1 Sequential Read Workload

As mentioned in the experimental setup, the logical block device of the guest VM can be represented as either a flat file or a physical raw disk partition at the hypervisor level. However, the different representation of the guest VM's block device directly affects the number of I/O requests served at the hypervisor level. For the selected combinations of **Ext3/JFS** and **Ext3/BD**, as Figure 9 shows, the number of I/O requests served at the hypervisor's block layer is significantly lower than that at the guest's block layer. More specifically, if JFS is used as a host file system, it greatly reduces the number of queued I/O requests sent from the guest level, resulting in much fewer I/O requests served at the hypervisor level than those at the guest level. If a raw disk partition is used instead, although there is no reduction on the number of queued I/O requests, the hypervisor level's block layer also lowers the number of served I/O requests by merging queued I/O requests.

There are two root causes for these I/O behaviors: (1) the file prefetching technique at the hypervisor level, known as *readahead*, and (2) the merging activities at the hypervisor level introduced by the I/O scheduler. The detailed descriptions of these root causes are given below.

First, there are frequent accesses to both files' content and metadata in a sequential read dominated workload. To expedite this process, readahead I/O requests are issued at the kernel level of both the guest and the hypervisor. Basically, readahead I/O requests populate the page cache with data already read from the block device, so that subsequent reads from the accessed files do not block on other I/O requests. As a result, it decreases the number of accesses to the block device. In particular, at the hypervisor level, a host file system issues readahead

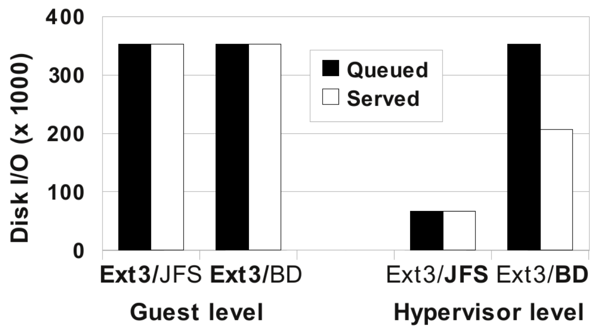


Figure 9: Disk I/Os under sequential read workload

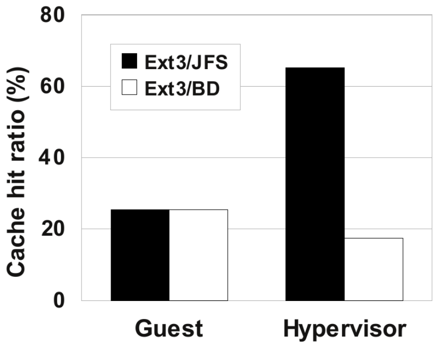


Figure 10: Cache hit ratio under sequential read workload.

requests and attempts to minimize the frequent accesses on the flat file by caching the subsequently accessed contents and metadata in the physical memory. Therefore, the I/Os served at the hypervisor level are much fewer than those at the guest level.

However, when accessing a raw disk partition, there is no readahead. Thus, for sequential workloads, a host file system outperforms a raw disk partition due to more effective caching. This discrepancy of data caching at the hypervisor level is clearly shown in Figure 10.

Second, to optimize I/O requests being served on the block device, the hypervisor's block layer attempts to reduce the number of accesses into the block device by sorting and merging queued I/O requests. However, when many I/O requests are sorted and merged, they need to stay longer in the queue than normal. For JFS (host file system), as shown in Figure 9, due to the effective caching, much fewer I/O requests are sent to the disk, and thus much fewer sorting/merging activities occur at the I/O queue. However, when a raw partition is used, much more I/O requests need to be sorted/merged. The sorting/merging activities cause a higher idle time (I2D) for I/O requests being served on the block device than those on the JFS (host file system). This behavior is depicted in Figure 11 (hypervisor level).

Remark: When a flat file is used as a guest VM's logical block device, sequential read dominated workloads

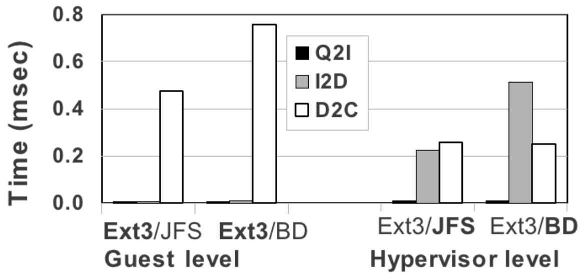


Figure 11: I/O times under sequential read workload.

can take advantage of the readahead at the hypervisor, achieving effective data caching. In contrast, when a disk partition is used, there is no readahead and data caching. Therefore, for all file systems, to gain high I/O performance, we recommend cloud administrators to select a flat file over raw partitions for services dominated by sequential reads.

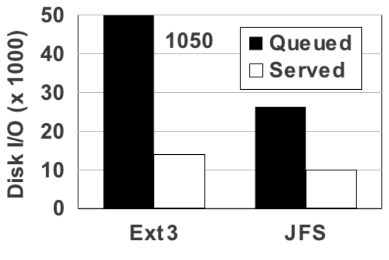
4.3.2 Sequential Write Workload

Our investigation uncovers the root causes of the nested file systems' performance dependency under a sequential write workload in two cases: (A) two file system combinations hold the same host file system, and (B) two combinations hold the same guest file system. The analysis detailed below focuses on two principal factors: sensitivity of an I/O scheduler and effectiveness of block allocation mechanisms.

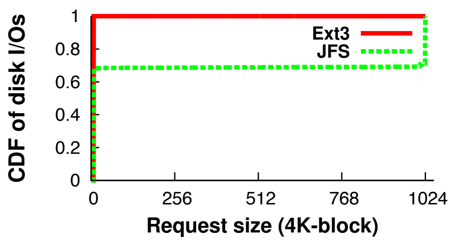
A. Different guests (Ext3, JFS) on the same host (ReiserFS) As shown in Figure 8 (B), we can see that the I/O performance of Ext3/ReiserFS is much worse than that of Ext3/BD, while the I/O performance of JFS/ReiserFS is much better than JFS/BD. At the guest level, we analyze the performance dependency of Ext3 and JFS based on the comparison of their I/O characteristics. The details of this comparison are shown in Figure 13.

Figure 13 (A) shows that most I/Os issued from Ext3 and sent to the block layer are well merged at the guest level's I/O scheduler. The effective merging of I/Os significantly reduces the number of I/Os to be served on Ext3 (guest). Meanwhile, Figure 13 (B) shows that 99% I/Os of Ext3 are in small size (8K) and those of JFS is 68%. Apparently, merging multiple small size I/Os incurs additional overhead. This is because the small requests have to be waited longer in the queue in order to be merged, thus, increasing their idle times. This behavior is illustrated in Figure 13 (C).

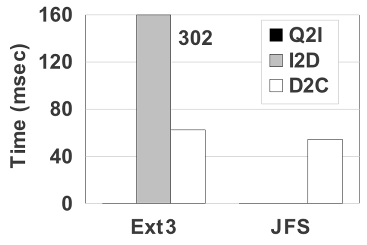
To understand the root cause of merging happened on Ext3 and JFS (guest), we perform a deep analysis by monitoring every issued I/O activities at the guest level.



A

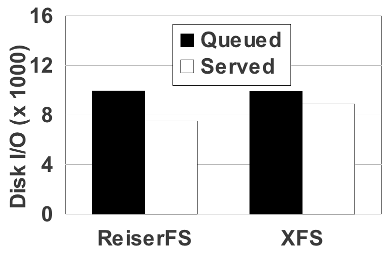


B

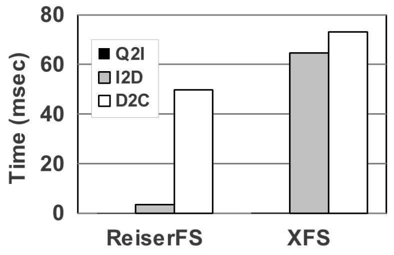


C

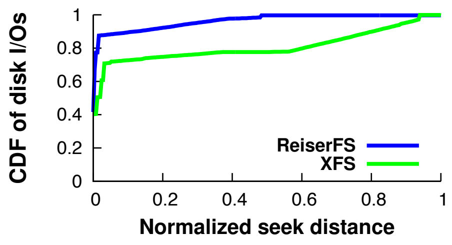
Figure 13: I/O characteristics at **guest level**: (A) disk I/Os, (B) I/O size, and (C) average I/O time.



A



B



C

Figure 14: I/O characteristics at **hypervisor level**: (A) disk I/Os, (B) average I/O time, and (C) disk seeks.

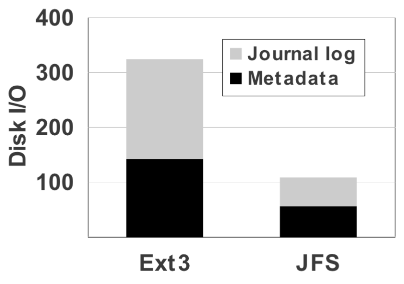


Figure 12: Extra I/O for journal log and metadata updates under sequential write workload.

What we found is that the block allocation mechanism causes this performance variation. To minimize disk seeks, Ext3 issues I/Os to allocate blocks of data on disk close to each other. The data includes regular data file, its metadata, and journal logs of metadata. This allocation scheme makes most I/Os be *back* merged. A back merge behavior denotes that a new request sequentially falls behind an exiting request on an order of the start sector, as they are logically adjacent. Note that two I/Os are logically adjacent when the end sector of one I/O is logically located next to the begin sector of the other I/O. As we can see, clustering adjacent I/Os facilitates the data access. However, it requires the issued I/Os to be waited longer in the queue for being processed.

JFS is more efficient than Ext3 in journaling. For regular data file written into disk, both Ext3 and JFS effectively coalescence multiple write operations to reduce the number of I/O committed into disk. However, for metadata and journal logs, instead of independently committing every single concurrent log entry as Ext3, JFS requires multiple concurrent log entries to be coalesced as one commit. For this reason, as shown in Figure 12, JFS has less I/Os spent for journaling, resulting in less performance degradation.

Remarks: The efficiency provided by the I/O scheduler's optimization is no longer valid for all nested file systems. Since file systems allocate blocks on disk differently, nested file systems have different impacts on performance when one particular I/O scheduler is used. Therefore, a nested file system should be chosen based on the effectiveness of underlying I/O scheduler's operations on its block allocation scheme.

B. Same guest (JFS) on different hosts (ReiserFS, XFS) Based on results of sequential writes shown in Figure 8 (B), JFS (guest) performs better on ReiserFS than on XFS. We analyze I/O activities of these host file systems to uncover differences of their block allocation mechanisms. The detailed analysis is given below.

The analysis of I/O activities reveals that the I/O scheduler processes ReiserFS' I/Os similarly to those of XFS. As shown in Figure 14 (A), the number of host file systems' I/Os to be queued and served are fairly similar in ReiserFS and XFS. However, Figure 14 (B) denotes that XFS' I/Os are executed slower than those of ReiserFS. A further analysis is needed to explain this behavior. In general, file systems allocate blocks on disk differently, thus, resulting in a different execution time for I/Os. For this reason, we perform an analysis on the disk seeks. Based on the results shown in Figure 14 (C), we find that long distance disk seeks on XFS cause high overhead and reduce its I/O performance. Note that in

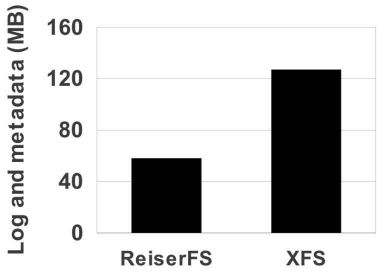


Figure 15: Extra data written into disk under the same workload from JFS (**guest**).

Figure 14 (C), the x-axis is represented as a normalized seek distance and 1 denotes the longest seek distance of the disk head, from one end to the other end of the partition.

With respect to the case of one host file system allocates disk blocks more effectively than another under the same workload, we analyze the mechanisms to allocate disk blocks of ReiserFS and XFS and find that XFS induces an overhead because of a multiple journal logging. The detailed explanations are as follows:

A multiple logging mechanism of metadata also incurs an overhead on XFS. Basically, XFS is able to record multiple separate changes occurred on the metadata of a single file and store them into journal logs. This technique effectively avoids such changes to be flushed into disk before another new change will be logged. However, every change of metadata can be range from 256 Bytes to 2 KB in size, while the default size of the log buffer is only 32 KB. Under an intensive write dominated workload, this small log buffer causes multiple changes of the file metadata to be frequently logged. As shown in Figure 15, this repeatedly logging produces extra data written into disk, thus, resulting in a performance loss.

Remarks: (1) An effective block allocation of one particular file system no longer guarantees a high performance when it runs on top of another file system. (2) Under an intensive write dominated workload, an update of journal logs on disk should be carefully considered to avoid performance degradation. Especially for XFS, the majority of its performance loss is attributed to not only a placement of journal logs, but also a technique to handle updates of these logs.

5 Discussion

Despite various practical benefits in using nested file systems in a virtualized environment, our experiments have shown the associated performance overhead to be significant if not configured properly. Here we offer five advice on choosing the *right* guest/host file system configurations to minimize performance degradation, or in some cases, even improve performance.

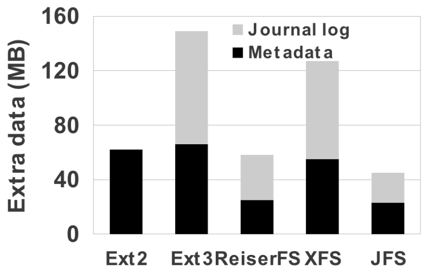


Figure 16: (**hypervisor level**) Extra data written into disk under a write-dominated workload from guest VM.

Advice 1 For workloads that are read-dominated (both sequential and random), using nested file systems has minimal impact on I/O throughput, independent of guest and host file systems. For workloads that have a significant amount of sequential reads, nested file systems can even improve throughput due to the readahead mechanism at the host level.

Advice 2 On the other hand, for workloads that are write-dominated, one should avoid using nested file systems in general due to i) one more layer to pass through and ii) additional metadata update operations. If one must use nested file systems, journaled file systems in the host should be avoided. Journaling of both metadata and data can cause significant performance degradation, and therefore, is not practical to use for most workloads, and if only metadata is journaled, a crash can corrupt a VM image file easily, thus, giving no benefit to metadata-only journaling mode in the host. As shown in Figure 16, the additional metadata writes to the journal log can result in significantly more I/O traffic. Performance is even more impacted if the location of the log is placed far away from either the metadata or the data locations.

Advice 3 For workloads that are sensitive to I/O latency, one should also avoid using nested file systems. As shown in Figure 6, even in the best case scenarios, nested file systems could increase I/O latency by 10-30% due to having an additional layer of file system to traverse and one more I/O queue to wait for.

Advice 4 In a nested file system, data and metadata placement decisions are made twice, first in the guest file system and then in the host file system. Guest file system uses various temporal and spatial heuristics to place related metadata and data blocks close to each other. However, when these placement decisions reach the host file system, it can no longer differentiate between data and metadata and treats everything as data. As a result, the secondary data placement decisions made by a host file system are both unnecessary and less efficient than those made by a guest file system. Ideally, the host file system should simply act as a pass-through layer such as VirtFS [22].

Advice 5 In our experiments, we used the default set of formatting and mounting parameters in all the file systems. However, just like in a non-virtualized environment, these parameters can be tuned to improve performance. There are more benefits in tuning the host file system's parameters than guest's as it is ultimately the layer that communicates with the storage device.

One should tune its parameters in such a way that the host file system most resembles a "dumb" disk. For example, when a disk is instructed to read a small disk block, it will actually read the entire track or cylinder and keep them in its internal cache to minimize mechanical movement for future I/O requests. A host file system can emulate this behavior by using larger block sizes.

Metadata operations at host file system is another source of overhead. When a VM image file is accessed or modified, its metadata often has to be modified, thus, causing additional I/O load. Parameters such as *noatime* and *nodiratime* can be used to avoid updating the last access time without losing any useful information. However, when the image file is modified, there is no option to avoid updating the metadata. As the image file will stay constant in size and ownership, the only field in the metadata that needs to be updated is the last modified time, which for an image file is just pure overhead. Perhaps this can be implemented as a file system mount option. Note that journaling, as mentioned previously, in the metadata-only mode has very little usage in the host level.

Lastly, using more advanced file system features to configure block groups and B+ trees to perform intelligent data allocation and balancing tasks will most likely be counter-productive. This is because these features will cause guest file system's view of disk layout to deviate further from the reality.

6 Conclusion

Our main objective is to better understand performance implications when file systems are nested in a virtualized environment. The major finding is that the choice of nested file systems on both hypervisor and guest levels has a significant performance impact on I/O performance. Traditionally, a guest file system is chosen based on the anticipated workload, regardless of the host file system. By examining a large set of different combinations of host and guest file systems under various workloads, we have demonstrated the significant dependency of the two layers on performance, and hence, system administrators must be careful in choosing *both* file systems in order to reap the greatest benefit from virtualization. In particular, if workloads are sensitive to I/O latency, nested file systems should be avoided or host file systems should simply perform as a pass-through layer in certain cases.

The intricate interactions between host and guest file systems represent an exciting and challenging optimization space for improving I/O performance in virtualized environments. Our preliminary investigation on nested file systems will help researchers to better understand critical performance issues in this area, and shed light on finding more efficient methods in utilizing virtual storage. We hope that our work will motivate system designers to more carefully analyze the performance gap at the real and virtual boundaries.

Acknowledgements

We are grateful to the anonymous referees and our shepherd, Andrea Arpaci-Dusseau, for their detailed feedback and guidance. This work was partially supported by NSF grant 0901537 and ARO grant W911NF-11-1-0149.

References

[1] Amazon Elastic Compute Cloud - EC2. `http://aws.amazon.com/ec2/` [Accessed: Sep 2011].

[2] blktrace - generate traces of the I/O traffic on block devices. `git://git.kernel.org/pub/scm/linux/kernel/git/axboe/blktrace.gitbt` [Accessed: Sep 2011].

[3] Filebench. `www.solarisinternals.com/wiki/index.php/FileBench` [Accessed: Sep 2011].

[4] FIO - Flexible I/O Tester. `http://freshmeat.net/projects/fio` [Accessed: Sep 2011].

[5] IBM Ccloud Computing. `http://www.ibm.com/ibm/cloud/` [Accessed: Sep 2011].

[6] Nested svm virtualization for kvm. `http://avikivity.blogspot.com/2008/09/nested-svm-virtualization-for-kvm.html` [Accessed: Sep 2011].

[7] The QCOW2 Image Format. `http://people.gnome.org/~markmc/qcow-image-format.html` [Accessed: Sep 2011].

[8] VirtualBox VDI. `http://forums.virtualbox.org/viewtopic.php?t=8046` [Accessed: Sep 2011].

[9] VMware Tools for Linux Guests. `http://www.vmware.com/support/ws5/doc/ws_newguest_tools_linux.html` [Accessed: Sep 2011].

[10] VMWare Virtual Disk Format 1.1. `http://www.vmware.com/technical-resources/interfaces/vmdk.html` [Accessed: Sep 2011].

[11] Window Azure - Microsoft's Cloud Services Platform. `http://www.microsoft.com/windowsazure/` [Accessed: Sep 2011].

[12] Xen Hypervisor Source. `http://xen.org/products/xen_archives.html` [Accessed: Sep 2011].

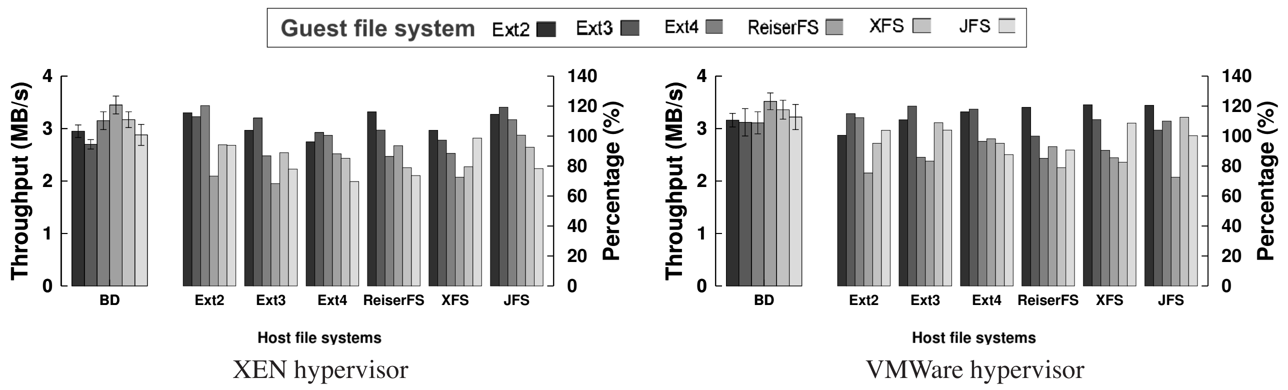


Figure 17: Other hypervisors show variation of relative **I/O throughput** of guest file systems under database workload (**higher is better**)

[13] Xen source - progressive paravirtulization. `http://xen.org/files/summit_3/xen-pv-drivers.pdf` [Accessed: Sep 2011].

[14] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX ATC'05*, April 2005.

[15] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *USENIX OSDI'10*, October 2010.

[16] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis. BORG: Block-reORGanization for Self-optimizing Storage Systems. In *USENIX FAST'09*, February 2009.

[17] D. Boutcher and A. Chandra. Does virtualization make disk scheduling passé? In *USENIX HotStorage'09*, October 2009.

[18] L. Cherkasova, D. Gupta, and A. Vahdat. When virtual is harder than real: Resource allocation challenges in virtual machine based IT environments, Feburary 2007.

[19] K. J. Duda and D. R. Cheriton. Borrowed-virtual-time (bvt) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, SOSP '99, Charleston, SC, USA, 1999.

[20] H. Huang, W. Hung, and K. G. Shin. FS2: dynamic data replication in free disk space for improving disk performance and energy consumption. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, Brighton, United Kingdom, 2005.

[21] K. Huynh and S. Hajnoczi. KVM/QEMU Storage Stack Performance Discussion. In *Proposals of Linux Plumbers Conference*, Cambridge, MA, USA, November 2010.

[22] V. Jujjuri, E. V. Hensbergen, and A. Liguori. VirtFS - A virtualization aware File System pass-through. In *Proceedings of the Ottawa Linux Symposium*, 2010.

[23] M. Kesavan, A. Gavrilovska, and K. Schwan. On Disk I/O Scheduling in Virtual Machines. In *USENIX WIOV'10*, Pittsburgh, PA, USA, March 2010.

[24] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, 2007.

[25] D. Ongaro, A. L. Cox, and S. Rixner. Scheduling I/O in virtual machine monitors. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '08, Seattle, WA, USA, 2008.

[26] R. Russell. virtio: towards a de-facto standard for virtual I/O devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, 2008.

[27] S. R. Seelam and P. J. Teller. Virtual I/O scheduler: a scheduler of schedulers for performance virtualization. In *ACM VEE'07*, June 2007.

[28] P. J. Shenoy and H. M. Vin. Cello: A Disk Scheduling Framework for Next Generation Operating Systems. In *Proceedings of ACM SIGMETRICS Conference*, 1997.

[29] S. Sivathanu, L. Liu, M. Yiduo, and X. Pu. Storage Management in Virtualized Cloud Environment. *IEEE Cloud Computing'10*, 2010.

[30] K. Suzaki, T. Yagi, K. Iijima, N. A. Quynh, and Y. Watanabe. Effect of readahead and file system block reallocation for lbcas. In *Proceedings of the Linux Symposium*, July 2009.

[31] C. Tang. Fvd: a high-performance virtual machine image format for cloud. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, Portland, OR, 2011.

[32] B.-A. Yassour, M. Ben-Yehuda, and O. Wasserman. On the DMA mapping problem in direct device assignment. In *SYSTOR'10: The 3rd Annual Haifa Experimental Systems Conference*, Haifa, Israel, May 2010.

Appendix

We have conducted experiments with the database workload to verify if the I/O performance of nested file systems is hypervisor-dependent. The chosen hypervisors are architecturally akin to KVM, such as VMware Player 3.1.4 with guest tools [9], and Xen 4.0 with Xen para-virtualized device drivers [12]. Figure 17 shows that the I/O performance variations of guest file systems on Xen and VMware are fairly similar to those on KVM.

Consistency Without Ordering

Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
Computer Sciences Department, University of Wisconsin, Madison

Abstract

Modern file systems use ordering points to maintain consistency in the face of system crashes. However, such ordering leads to lower performance, higher complexity, and a strong and perhaps naive dependence on lower layers to correctly enforce the ordering of writes. In this paper, we introduce the No-Order File System (NoFS), a simple, lightweight file system that employs a novel technique called backpointer-based consistency to provide crash consistency without ordering writes as they go to disk. We utilize a formal model to prove that NoFS provides data consistency in the event of system crashes; we show through experiments that NoFS is robust to such crashes, and delivers excellent performance across a range of workloads. Backpointer-based consistency thus allows NoFS to provide crash consistency without resorting to the heavyweight machinery of traditional approaches.

1 Introduction

One of the core problems in file systems research over the years has been the challenge of providing consistency in the presence of system crashes. There have been a number of solutions to tackle this problem: from the simple file-system check [20] of the Fast File System [18] to the complicated copy-on-write mechanism of ZFS [3]. Each approach has a different core technique: write-ahead logging [12], copy-on-write [15] or tracking dependencies among writes to disk [10].

Although these approaches all differ vastly in their details, they share one common trait: each uses a careful ordering of writes to implement its update protocol. Journaling file systems require that metadata and data are persisted before the commit record is written [2, 31, 41, 45]. Copy-on-write file systems require that the root block be updated only after the rest of the update is safely on disk [15, 32, 40, 48]. Soft updates is built entirely around the careful ordering of disk writes [10].

In the event of a crash, ordering points allow the file system to reason about which writes reached the disk and which did not, enabling the file system to take corrective measures, such as replaying the writes, to recover. Unfortunately, ordering points are not without their own set of problems. By their very nature, ordering points introduce waiting into the file-system code, thus potentially lowering performance. They constrain the scheduling of disk writes, both at the operating system level and at the disk driver level. They introduce complexity into the file-system code, which leads to bugs and lower reliability [25, 26, 49, 50]. The use of ordering points also forces file systems to ignore the end-to-end argument [34], as the support of lower-level systems and disk firmware is required to implement imperatives such as the disk cache flush. When such imperatives are not properly implemented [36], file-system consistency is compromised [29]. In today's cloud computing environment [1], the operating system runs on top of a tall stack of virtual devices, and only one of them needs to neglect to enforce write ordering [47] for file-system consistency to fail.

We can thus summarize the current state of the art in file-system crash consistency as follows. At one extreme is a lazy, optimistic approach that writes blocks to disks in any order (e.g., ext2 [4]); this technique does not add overhead or induce extra delays at run-time, but requires an expensive (and often prohibitive) disk scan after a crash. At the other extreme are eager, pessimistic approaches that carefully order disk writes (e.g., ZFS or ext3); these techniques pay a perpetual performance penalty in return for consistency guarantees and quick recovery. We seek to obtain the best of both worlds: the simplicity and performance benefits of the lazy approach with the strong consistency and availability of eager file systems.

We present the *No-Order file system* (NoFS), a simple, optimistic, lightweight file system which maintains consistency without resorting to the use of ordering. NoFS employs a new approach to providing consistency called *backpointer-based consistency*, which is built upon references in each file-system object to the files or directories that own it. We extend a logical framework for file systems [38] to prove that the incorporation of backpointer-based consistency in an order-less file system guarantees a certain level of consistency. We simplify the update protocol through *non-persistent allocation structures*, reducing the number of blocks that need to reach disk to successfully complete an operation.

Through reliability experiments, we demonstrate that NoFS is able to detect and handle a wide range of inconsistencies. We compare the performance of NoFS with ext2, an order-less file system with no consistency guarantees, and ext3, a journaling file system with metadata consistency. We show that NoFS has excellent performance overall, matching or exceeding the performance of ext2 and ext3 on various workloads. We also discuss the limitations of our approach.

2 Background

File systems use a number of data structures to keep track of the data on disk. These include allocation structures such as bitmaps, and metadata such as inodes. In order to do a single operation such as file creation, multiple data structures have to be updated on disk. For example, in the ext2 file system [4], in order to create an empty file, the inode bitmap, the parent inode, the parent directory, and the child inode all need to be updated and written to disk.

The problem of file-system consistency arises because the system may crash at any time, resulting in some of the updates persisting, and other updates being lost. File-system inconsistency manifests in different ways: a missing file, a file with garbage data, or in some cases, an unmountable file system. File systems have different solutions to this problem, with varying levels of consistency.

We first examine the different levels of consistency provided by file systems, describing the guarantees provided by each level. We then examine the techniques used in file systems to provide consistency and show that all of them (except the file-system check) have at least one ordering point in their update protocols. We discuss the disadvantages of having ordering points and motivate the design of our order-less file system.

2.1 File-system consistency

There are many levels of consistency in file systems, differing in terms of guarantees provided for data and metadata blocks. An inconsistency could be caused by many things: a hardware error, memory corruption, or a system crash. In this work, we are only concerned with inconsistencies occurring due to a system crash.

Metadata consistency: The metadata structures of the file system are entirely consistent with each other. There are no dangling files and no duplicate pointers. The counters and bitmaps of the file system, which keep track of resource usage, match with the actual usage of resources on the disk. Therefore a resource is in use if and only if the bitmaps say that it is in use. Metadata consistency does not provide any guarantees about data.

Data consistency: Data consistency is a stronger form of metadata consistency. Along with the guarantee about metadata, there is the additional guarantee that all data that is read by a file belongs to that file. In other words, a read of file A may not return garbage data, or data belonging to some file B. It is possible that the read may return an older version of the data of file A.

Version consistency: Version consistency is a stronger form of data consistency with the additional guarantee that the metadata version matches the version of the referred data. For example, consider a file with a single data block. The data block is overwritten, and a new block is added, thereby changing the file version: the old version had one block, and the new version has two blocks. Version consistency guarantees that a read of the file does not return old data from the first block and new data from the second block (since the read would return the old version of the data block and the new version of the file metadata).

2.2 Techniques for providing consistency

In this section, we review different approaches to providing consistency in file systems. We point out where ordering points are needed in each of the techniques, except for file-system checks. An ordering point signifies that some blocks need to be persistent on disk before other blocks. For example, an update protocol might require that all the file-system metadata reach the disk before all the data.

2.2.1 File-system check

The file-system check is the simplest solution to the consistency problem: let the system crash and become inconsistent, and upon reboot, fix the inconsistencies. This technique was used in the Fast File System [18, 20] and the ext2 file system [4]. No extra actions are required during runtime, allowing the file system to execute without any performance degradation. The simplicity comes with a high cost: the entire disk needs to be scanned before inconsistencies can be fixed in the file system. While this was acceptable for early file systems that were megabytes in size, scanning an entire disk (or worse, a large RAID volume [23]) would require hours in modern systems. Though several optimizations were developed to reduce the running time of the file-system check [13, 19, 24], it is still too expensive for large volumes, prompting the file-system community to turn to other solutions.

File systems that depend upon on the file-system check alone for consistency cannot provide data consistency, since there is no way for the file system to differentiate between valid data and garbage in a data block. Therefore file reads may return garbage after a crash. The state of every metadata structure is known after the disk scan, and hence duplicate resource allocation and orphan resources can be handled, ensuring metadata consistency.

2.2.2 Journaling

Journaling uses the idea of write-ahead logging [12] to solve the consistency problem: metadata (and sometimes data) is first logged to a separate location on disk, and when all writes have safely reached the disk, the information is written into its original place in the file system. Over the years, this technique has been incorporated into a number of file systems such as NTFS [21], JFS [2], XFS [41], ReiserFS [31], and ext3 [45, 46].

Journaling file systems offer data or metadata consistency based on whether data is journaled or not. Both journaling modes use at least one ordering point in their update protocols, where they wait for the journal writes to be persisted on disk before writing the commit block. Journaling file systems often perform worse than their order-less peers, since information needs to be first written to the log and then later to the correct location on disk. Recovery of the journal is needed after a crash, but it is usually much faster than the file-system check.

2.2.3 Soft updates

Soft updates involves tracking dependencies among in-memory copies of metadata blocks, and carefully ordering the writes to disk such that the disk always sees content that is consistent with the other disk metadata. In order to do this, it may sometimes be necessary to roll back updates to a block at the time of write, and roll-forward the update later. Soft updates was implemented for FFS, and enabled FFS to achieve performance close to that of a memory-based file system [10] . However, it was extremely tricky to implement the ordering rules correctly, leading to numerous bugs. Though the Featherstitch project [9] reduces the complexity of soft updates, the idea has not spread beyond the BSD distributions.

Soft updates provide metadata and data consistency at low cost. FFS with soft updates cannot tell the difference between different versions of data, and hence does not provide version consistency. Soft updates also provide high availability since a blocking file-system check is not required; instead, upon reboot after a crash, a snapshot of the file-system state is taken, and the file-system check is run on the snapshot in the background [19].

2.2.4 Copy-on-write

The copy-on-write technique, as the name suggests, directs a write to a metadata or data block to a new copy of the block, never overwriting the block in place. Once the write is persisted on disk, the new information is added to the file-system tree. The ordering point is in-between these two steps, where the file system atomically changes between the old view of the metadata to one which includes the new information. Copy-on-write has been used in a number of file systems [15, 32], with the most recent being ZFS [3] and btrfs [48].

Copy-on-write file systems provide metadata, data, and version consistency due to the use of logging and transactions. Modern copy-on-write file systems like ZFS achieve good performance, though at the cost of very high complexity. The large size of these file systems (tens of thousands of lines of code [35]) is partly due to the copy-on-write technique, and partly due to advanced features such as storage pools and snapshots.

2.3 Summary

Table 1 compares consistency techniques on complexity, performance, availability, and consistency guarantees provided. Observe that every technique that provides consistency and availability in file systems uses ordering points in its update protocol. Ordering points lead to complexity in the file-system code, paving the way for bugs and decreased reliability. File systems which use ordering points perform worse than order-less file systems on some workloads. The use of ordering points is built upon lower-level functionality such as the SATA flush command [43]; when disks do not reliably flush their cache [36], ordering points fail to enforce consistency and more complicated measures have to be taken [29]. Thus there is a need for a technique which provides consistency without sacrificing simplicity, availability, or performance. We believe that backpointer-based consistency fulfills this need.

	Consistency					
Technique	**Metadata**	**Data**	**Version**	**Complexity**	**Performance**	**Availability**
File-system check	√	×	×	L	H	L
Metadata journaling	√	×	×	M	M	H
Data journaling	√	√	√	M	M	H
Soft Updates	√	√	×	H	H	H
Copy-on-write	√	√	√	H	H	H
BBC	√	√	×	L	H	H

Table 1: **Consistency techniques.** *The table compares various approaches to providing consistency in file systems. Legend: L – Low, M – Medium, H – High. We observe that only backpointer-based consistency (BBC) provides data consistency with low complexity, high performance, and high availability.*

3 Design

We present the design of the *No-Order file system (NoFS)*, a lightweight, consistent file system with no ordering points in its update protocol. NoFS provides access to files immediately upon mounting, with no need for a file-system check or journal recovery.

In this section, we introduce *backpointer-based consistency (BBC)*, the technique used in NoFS for maintaining consistency. We use a logical framework to prove that BBC provides data consistency in NoFS. We discuss how BBC can be used to detect and recover from inconsistencies, and elaborate on why allocation structures are not persisted to disk in NoFS.

3.1 Overview

The main challenge in NoFS is maintaining consistency without ordering points. Consistency is closely tied to logical identity in file systems. Inconsistencies arise due to confusion about an object's identity; for example, two files may each claim to own a data block. If the block's true owner is known, such inconsistencies could be resolved. Associating each object with its logical identity is the crux of the backpointer-based consistency technique.

Employing backpointer-based consistency allows NoFS to detect inconsistencies on-the-fly, upon user access to corrupt files and directories. The presence of a corrupt file does not affect access to other files in any way. This property enables immediate access to files upon mounting, avoiding the downtime of a file-system check or journal recovery. A read is guaranteed to never return garbage data, though stale data may be returned.

We intentionally avoided using complex rules and dependencies in NoFS. We simplified the update protocols, not persisting allocation structures to disk. We maintain in-memory versions of allocation structures and discover data and metadata allocation information in the background while the file system is running.

3.2 Backpointer-based consistency

Backpointer-based consistency is built around the logical identity of file-system objects. The logical identity of a data block is the file it belongs to, along with its position inside the file. The logical identity of a file is the list of directories that it is linked to. This information is embedded inside each object in the form of a *backpointer*. Upon examining the backpointer of an object, the parent file or directory can be determined instantly. Blocks have only one owner, while files are allowed to have multiple parents. Figure 1 illustrates how backpointers link file-system objects in NoFS. As each object in the file system is examined, a consistent view of the file-system state can be incrementally built up.

Though conceptually simple, backpointers allow detection of a wide range of inconsistencies. Consider a block that is deleted from a file, and then assigned to another file and overwritten. If a crash happens at any point during these operations, some subset of the data structures on disk may not be updated, and both files may contain pointers to the block. However, by examining the backpointer of the block, the true owner of the block can be identified.

In designing NoFS, we assume that the write of a block along with its backpointer is atomic. This assumption is key to our design, as we infer the owner of the data block by examining the backpointer. Current SCSI drives allow a 520-byte atomic write to enable checksums along with each 512-byte sector [42]; we envision that future drives with 4-KB blocks will provide similar functionality.

Backpointers are similar to checksums in that they verify that the block pointed to by the inode actually belongs to the inode. However, a checksum does not identify the owner of a data block; it can only confirm that the correct block is being pointed to. Consistency and recovery require identification of the owner.

3.2.1 Intuition

We briefly provide some intuition about the correctness of using the backpointer-based consistency technique to ensure data consistency. We first consider what data consistency and version consistency mean, and the file-system structures required to ensure each level of consistency.

Data consistency provides the guarantee that all the data accessed by a file belongs to that file; it may not be garbage data or belong to another file. This guarantee is obtained when a backpointer is added to a data block.

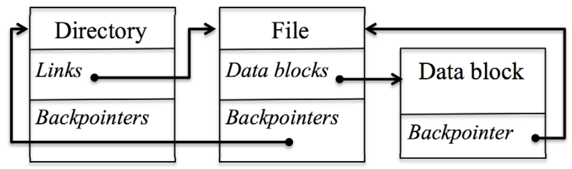


Figure 1: **Backpointers.** *The figure shows a conceptual view of the backpointers present in NoFS. The file has a backpointer to the directory that it belongs to. The data block has a backpointer to the file it belong to. Files and directories have many backpointers while data blocks have a single backpointer.*

Consider a file pointing to a data block. Upon reading the data block, the backpointer is examined. If the backpointer matches the file, then the data block must have belonged to the file, since the backpointer and the data inside the block were written together. If the data block was reallocated to another file and written, it would be reflected in the backpointer. Hence, no ordering is required between writes to data and metadata since the data block's backpointer would disagree in the event of a crash. Note that the data block could have belonged to the file at some point in the past; the backpointer does not provide any information about when the data block belonged to the file. Thus, the file might be pointing to an old version of the data block, which is allowed under data consistency.

Version consistency is a stricter form of data consistency which requires that in addition to belonging to the correct file, all accessed data must be the correct version. Stale data is not allowed in this model. Backpointers are not sufficient to enforce version consistency, as they contain no information about the version of a data block. Hence more information needs to be added to the file system. Each data block has a timestamp indicating when it was last updated. This timestamp is also stored in the inode containing the data block. When a block is accessed, the timestamp in the inode and data block must match. Since timestamps are a way to track versions, the versions in the inode and data block can be verified to be the same, thereby providing version consistency.

We decided against including timestamps in NoFS backpointers because updating timestamps in backpointers and metadata reduces performance and induces a considerable amount of storage overhead. Timestamps need to be stored with every object and its parent. Every update to an object involves an update to the parent object, the parent's parent, and so on all the way up to the root. Furthermore, doing so works against our goal of keeping the file system simple and lightweight; hence, NoFS provides data consistency, but not version consistency.

The full proof involves extending the logical framework of Sivathanu et al. [38] to prove that an order-less file system employing the backpointer-based consistency technique provides data consistency. We further prove that if the backpointer contains an update timestamp, the file system provides version consistency. The full proof can be found in the technical report [5].

3.2.2 Detection and Recovery

In NoFS, detection of an inconsistency happens upon access to corrupt files or data. When a data or metadata block is accessed, the backpointer is checked to verify that the parent metadata block has the same information. If a file is not accessed, its backpointer is not checked, which is why the presence of corrupt files does not affect access to other files: checking is performed on-demand.

This checking happens both at the file level and the data block level. When a file is accessed, it is checked to see whether it has a backpointer to its parent directory. This check allows identification of deleted files where the directory did not get updated, and files which have not been properly updated on disk.

NoFS is able to recover from inconsistencies by treating the backpointer as the true source of information. When a directory and a file disagree on whether the file belongs to the directory or not, the backpointer in the file is examined. If the backpointer to the directory is not found, the file is deleted from the directory. Issues involving blocks belonging to files are similarly handled.

3.3 Non-persistent allocation structures

In an order-less file system, allocation structures like bitmaps cannot be trusted after a crash, as it is not known which updates were applied to the allocation structures on disk at the time of the crash. Any allocation structure will need to be verified before it can be used. In the case of global allocation structures, all of the data and metadata referenced by the structure will need to be examined to verify the allocation structure.

Due to these complexities, we have simplified the update protocols in NoFS, making the allocation structures non-persistent. The allocation structures are kept entirely in-memory. NoFS starts out with empty allocation structures and allocation information is discovered in the background, while the file system is online. NoFS can verify whether a block is in use by checking the file that it has a backpointer to; if the file refers to the data block, the data block is considered to be in use. Similarly, NoFS can verify whether a file exists or not by checking the directories in its backpointers. Thus NoFS can incrementally learn allocation information about files and blocks.

4 Implementation

We now present the implementation of NoFS. We first describe the operating system environment, and then discuss the implementation of the two main components of NoFS: backpointers and non-persistent allocation structures. We describe the backpointer operations that NoFS performs for each file-system operation.

Action	Backpointer operations
Create	Write backlink into new inode
Read	Translate offset Verify block backpointer in data block
Write	Translate offset Verify block backpointer in data block
Append	Translate offset Write block backpointer into data block
Truncate	No backpointer operations
Delete	No backpointer operations
Link	Write backlink into inode
Unlink	Remove backlink from inode
mkdir	Write directory entry backpointer into directory block
rmdir	No backpointer operations

Table 2: **NoFS backpointer operations.** *The table lists the operations on backpointers caused by common file system operations. Note that all checks are done in memory.*

4.1 Operating system environment

NoFS is implemented as a loadable kernel module inside Linux 2.6.27.55. We developed NoFS based on ext2 file-system code. Since NoFS involves changes to the file-system layout, we modified the `e2fsprogs` tools 1.41.14 [44] used for creating the file system.

Linux file systems cache user data in a unified page cache [6]. File reads (except direct I/O) are always satisfied from the page cache. If the page is not up-to-date at the time of read, the page is first filled with data from the disk and then returned to the user. File writes cause pages to become dirty, and an I/O daemon called `pdflush` periodically flushes dirty pages to disk. Due to this tight integration between the page cache and the file system, NoFS involves modifications to the Linux page cache.

4.2 Backpointers

NoFS contains three types of backpointers. We describe each of them in turn, pointing out the objects they conceptually link, and how they are implemented in NoFS. Figure 2 illustrates how various objects are linked by different backpointers. Every file-system operation that involves the creation or access of a file, directory, or data block involves an operation on backpointers. These operations are listed in Table 2.

4.2.1 Block backpointers

Block backpointers are {*inode number, block offset*} pairs, embedded inside each data block in the file system. The first 8 bytes of every data block are reserved for the backpointer. Note that we need to embed the backpointer inside the data block since disks currently do not provide the ability to store extra data along with each 4K block atomically. The first 4 bytes denote the inode number of the file to which the data block belongs. The second 4 bytes represent the logical block offset of the data block

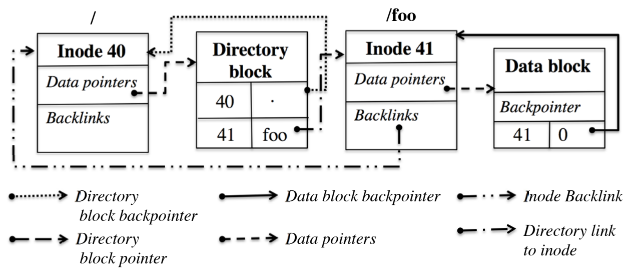


Figure 2: **Implementation of backpointers.** *The figure shows the different kinds of backpointers present in NoFS. foo is a child of the root inode /. This link is represented by a backlink from foo to /. Similarly, the data block is a part of foo, and hence has a backpointer to foo. Directory blocks also contain backpointers, in the form of dot entries to their owner's inode.*

within the file. Given this information, it is easy to check whether the file contains a pointer to the data block at the specified offset. Indirect blocks contain backpointers too, since they belong to a particular file. However, since the indirect block data is not logically part of a file, they are marked with a negative number for the offset.

Our implementation depends on the `read` and `write` system calls being used; data is modified as it is passed from the page cache to the user buffer and back during these calls. When these calls are by-passed (via `mmap`) or the page cache itself is by-passed (via direct IO mode), verifying each access becomes challenging and expensive. We do not support `mmap` or direct IO mode in NoFS.

Insertion: The data from a `write` system call goes through the page cache before being written to disk. We modified the page cache so that when a page is requested for a disk write, the backpointer is written into the page first and then returned for writing. The block offset translation was modified to take the backpointer into account when translating a logical offset into a block number.

Verification: Once a page is populated with data from the disk, the page is checked for the correct backpointer. If the check fails, an I/O error is returned. If this is the first time that the data block is accessed, the inode's attributes (size and number of blocks) are updated. Note that the page is not checked on every access, but only the first time that it is read from disk. Assuming memory corruption does not occur [51], this level of checking is sufficient.

4.2.2 Directory backpointers

The dot directory entry serves as the backpointer for directory blocks, as it points to the inode which owns the block. However, the dot entry is only present in the first directory block. We modified ext2 to embed the dot entry in every directory block, thus allowing the owner of any directory block to be identified using the dot entry.

Though the block backpointer could have been used in directory blocks as well, we did not do so for two reasons. First, the structured content of the directory block enables the use of the dot entry as the backpointer, simplifying our implementation. Second, the offset part of the block backpointer is unnecessary for directory blocks since directory blocks are unordered and appending a directory block at the end suffices for recovery.

Insertion: When a new directory entry is being added to the inode, it is determined whether a new directory block will be needed. If so, the dot entry in added in the new block, followed by the original directory entry.

Verification: Whenever the directory block is accessed, such as in `readdir`, the dot entry is cross-checked with the inode. If the check fails, an I/O error is returned and the directory inode's attributes (size and block count) are updated.

4.2.3 Backlinks

An inode's backlinks contain the inode numbers of all its parent directories. Every valid inode must have at least one parent. Hard linked inodes may have multiple parents.

We modified the file-system layout to add space for backlinks inside each inode. The inode size is increased from the default 128 bytes to 256 bytes, enabling the addition of 32 backlinks, each of size 4 bytes. The `mke2fs` tool was modified to create a backlink between the `lost+found` directory and the root directory when the file system is created.

Insertion: When a child inode is linked to a parent directory during system calls such as `create` or `link`, a backlink to the parent is added in the child inode.

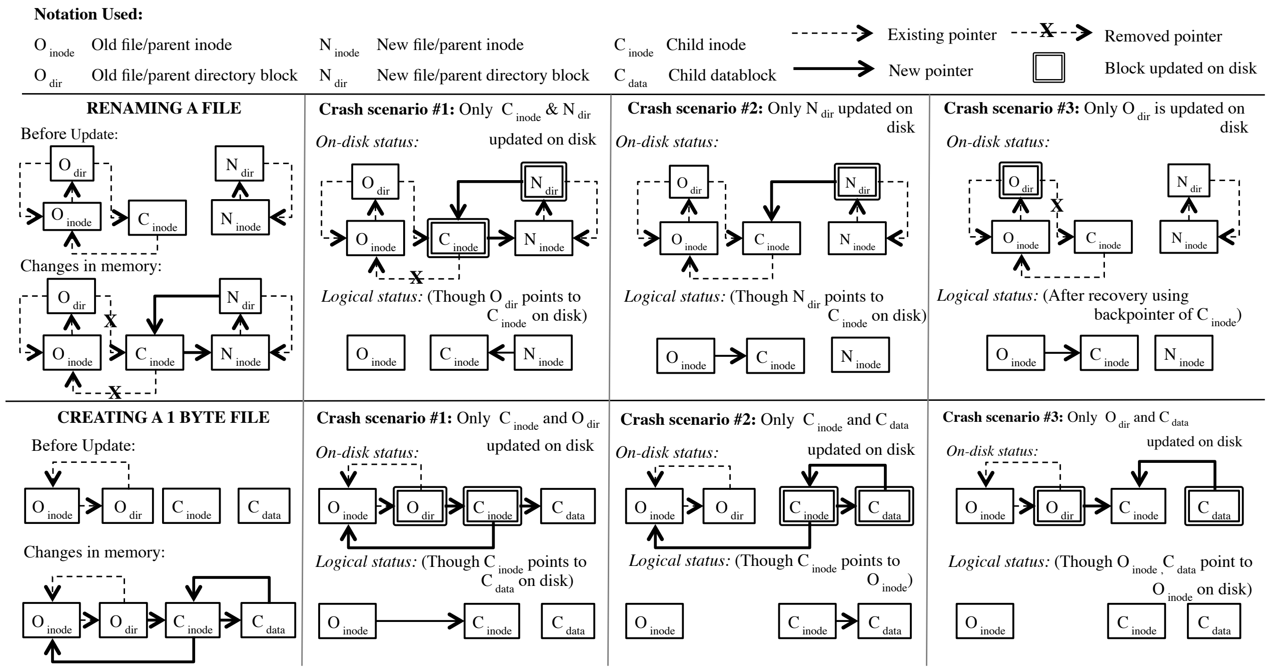
Verification: At each step of the iterative inode lookup process, we check that the child inode contains a backlink to the parent. A failed check stops the lookup process and returns an I/O error. If this is the first time the inode is accessed via this particular path, the number of links for the inode is updated.

4.2.4 Detection

Every data block is checked for a valid backpointer when it is read from the disk into the page cache. We assume that neither memory nor on-disk corruption happens; hence, it is safe to limit checking to when a data block is first brought into main memory. It is this property that leads to the high performance of NoFS; because disk I/O is several orders of magnitude slower than in-memory operations, the backpointer check can be performed on disk blocks with very low overhead.

Inode backlink checking occurs during directory path resolution. The child inode's backlink to the parent inode is checked. Since both inodes are typically in memory during directory path resolution, the backlink check is a quick in-memory check, and does not degrade performance significantly, since a disk read is not performed to obtain the parent or child inode.

Note that the detection of inconsistency happens at the level of a single resource, such as an inode or a data



Notation Used:
O inode Old file/parent inode
O dir Old file/parent directory block
N inode New file/parent inode
N dir New file/parent directory block
C inode Child inode
C data Child datablock
Existing pointer
New pointer
Removed pointer
Block updated on disk
RENAMING A FILE
Before Update:
Changes in memory:
Crash scenario #1: Only C inode & N dir updated on disk
On-disk status:
Logical status: (Though O dir points to C inode on disk)
Crash scenario #2: Only N dir updated on disk
Logical status: (Though N dir points to C inode on disk)
Crash scenario #3: Only O dir is updated on disk
Logical status: (After recovery using backpointer of C inode)
CREATING A 1 BYTE FILE
Crash scenario #1: Only C inode and O dir updated on disk
Logical status: (Though C inode points to C data on disk)
Crash scenario #2: Only C inode and C data updated on disk
Logical status: (Though C inode points to O inode)
Crash scenario #3: Only O dir and C data updated on disk
Logical status: (Though O inode, C data point to O inode on disk)

Figure 3: **Handling crashes with backpointers.** *The figure presents three failure scenarios during the rename of a file, and the creation of a file with 1 byte of data. In each scenario, employing backpointers allows us to detect inconsistencies such as both the old and new parents claiming the child, and the child pointing to a data block that hasn't been updated.*

block. Verifying that a data block belongs to an inode can be done without considering any other object in the file system. The presence of corrupt files or blocks does not affect the reads or writes to other non-corrupt files. As long as corrupt blocks are not accessed, their presence can be safely ignored by the rest of the system. This feature contributes to the high availability of NoFS: a file-system check or recovery protocol is not needed upon mount. Files can be immediately accessed, and any access of a corrupt file or block will return an error. This feature also allows NoFS to handle concurrent writes and deletes. Even if many writes and deletes were going on at the time of a crash, NoFS can still detect inconsistencies by considering each inode and data block pair in isolation.

Let us illustrate this with an example. Upon mount, we run the command `cat /dir1/file1`, which involves several checks in the file system. First, the directory block for `dir1` is fetched, and checked whether it has a directory backpointer to the root directory. Similarly, when the `file1` inode is retrieved from disk, it is checked to see if it has a backlink to `dir1`. When the data block of `file1` is retrieved, it is checked to verify that the data block has a block backpointer to `file1`. If any of these checks fail, an error is returned to the user.

Figure 3 illustrates the detection of inconsistencies during different crash scenarios for two operations: renaming a file and creating a single byte file. The state of data structures in memory before and after the update is first shown. In each crash scenario, a different subset of the in-memory updates is successfully written to disk. The state of various pointers on disk after the crash is shown, followed by the consistent logical view that NoFS obtains after verification using back pointers. For example, during the rename, a crash may lead to the file being listed in both the old and new directories. However, the logical status shows that upon backpointer verification, the true owner of the child inode is found using the backlink.

4.2.5 Recovery

Having backlinks and backpointers allows recovery of lost files and blocks. Files can be lost due to a number of reasons. A rename operation consists of a unlink and a link operation. An inopportune crash could leave the inode not linked to any directory. A crash during the create operation could also lead to a lost file. Such a lost file can be recovered in NoFS, due to the backlinks inside each inode. Each such inode is first checked for access to all its data blocks. If all the data blocks are valid, it is a valid subtree in the file system and can be inserted back into the directory hierarchy (using the backlinks information) without compromising the consistency of the file system. When adding a directory entry for the recovered inode, it is correct to append the directory entry at the end of the directory, since directory entries are an unordered collection; there is no meaning attached to the exact offset inside a directory block where a directory entry is added.

In a similar fashion, it it possible to recover data blocks lost due to a crash before the inode is updated. A data block, once it has been determined to belong to an inode, cannot be embedded at an arbitrary point in the inode data. It is for this reason that the *offset* of a data block is embedded in the data block, along with the inode number. The offset allows a data block to be placed exactly where it belongs inside a file. Indirect blocks of a file do not have the offset embedded, as they do not have a logical offset within the file. Indirect blocks are not required to reconstruct a file; only data blocks and their offsets are needed.

Using reconstruction of files from their blocks on disk, files can be potentially "undeleted", provided that the blocks have not been reused for another file. We have not implemented undelete in NoFS. Block allocation would need to be tweaked to not reuse blocks for a certain amount of time, or until a certain free-space threshold is reached. Undelete might turn up stale data because NoFS does not support version consistency; the data block might have been part of an older version of the inode.

4.3 Non-persistent allocation structures

The allocation structures in ext2 are bitmaps and group descriptors. These structures are not persisted to disk in NoFS. In-memory versions of these data structures are built using the *metadata scanner* and *data scanner*. Statistics usually maintained in the group descriptors, such as the number of free blocks and inodes, are also maintained in their in-memory versions.

Upon file-system mount, in-memory inode and block bitmaps are initialized to zero, signifying that every inode and data block is free. Since every block and inode has a backpointer, it can be determined to be in use by examining its backlink or backpointer, and cross-checking with the inode mentioned in the backpointer. As every object is examined, consistent file-system state is built up and eventually complete knowledge of the system is achieved.

In the file system, a block or inode that is marked free could mean two things: it is free, or it has not been examined yet. Since all blocks and inodes are marked free at mount time, inodes need to be examined to check that they are indeed free; hence blocks or inodes that have not been examined yet cannot be allocated. In order to mark which inodes or blocks have been examined, we added a new bitmap each for inodes and data blocks called the *validity* bitmap. If a block or inode has been examined and marked as free, it is safe to use it. Blocks not marked as valid could actually be used blocks, and hence must not be used for allocation. The examination of inodes and blocks are carried out by two background threads called the metadata scanner and data scanner. The two threads work closely together in order to efficiently find all the used inodes and blocks on disk.

4.3.1 Metadata Scan

Each inode needs to be examined in order to find out if it is in use or not. The backlinks in the inode are found, and the directory blocks of the referred inodes are searched for a directory entry to this inode. Note that the directory hierarchy is not used for for the scan. The disk order of inodes is used instead, as this allows for fast sequential reads of the inode blocks.

Once an inode is determined to be in use, its data blocks have to verified. This information is communicated to the data scanner by adding the data blocks of the inode to a list of data blocks to be scanned. The inode information is also attached to the list so that the data scanner can simply compare the backpointer value to the attached value to determine whether the block is used. However, if the inode has indirect blocks, the inode data blocks are explored and verified immediately. An inode with indirect blocks may contain thousands of data blocks, and it would be cumbersome to add all those data blocks to the list and process them later; hence inode data is verified immediately by the metadata scanner. Each inode is marked valid after it has been scanned, allowing inode allocation to occur concurrently with the metadata scan.

4.3.2 Data Scan

Observe that a data block is in use only if it is pointed to by a valid inode which is in use; hence only data blocks that belong to a valid inode need to be checked, which reduces the number of blocks that need to be checked drastically.

The data block scanner works off a list of data blocks that the metadata scanner provides. Each list item also includes information about the inode that contained the data block. Therefore, the data scanner simply needs to read the inode off the disk and compare the backpointer inode to the inode information in the list item. The data block is marked valid after the examination is complete.

Since the data scanner only looks at blocks referred to by inodes, there may be plenty of unexamined blocks which are not referred and potentially free. These blocks cannot be marked as valid and free until the end of the data scan, when all valid inodes have been examined. While the scan is running, the file system may indicate that there are no free blocks available, even if there are many free blocks in the system. In order to fix this, we implemented another scanner called the sequential block scanner which reads data blocks in disk order and verifies them one by one. This thread is only started if no free blocks are found, and the data scanner is still running.

4.4 Limitations

The design of NoFS involves a number of trade-offs. We describe the limitations that arise from our design choices.

Recovery: NoFS was designed to be as lightweight as possible, avoiding heavy machinery for logging or copy-on-write. As a result, file-system recovery is limited. For example, consider a file that is truncated, and later written with new data. After a crash in the middle of these updates, the file may point to a block that it does not own. This inconsistency is detected upon access to the data block. However, the version of the file which pointed to its old data cannot be recovered easily. By utilizing logging, a file system like ext3 provides the ability to preserve data in the event of a crash.

Transactions: NoFS does not provide atomic transactions. Operations can be partially applied to different data structures. For example, if the file system crashes in the middle of a rename, it is possible that the file appears both in the old and new directories, as we do not validate directory entries during a `readdir`. Though the user will be able to access the file via only one directory, the 'old-or-new' aspect of transactions is not provided.

Accessing unverified objects: For large disks, it is possible that an object is accessed before the scan has verified it. Accessing such unverified objects involves a performance cost. The performance cost is felt during different system calls for inodes and data blocks.

Running the `stat` system call on an unverified inode may result in invalid information, as the number of blocks recorded in the inode may not match the actual number of blocks that belong to the inode on disk. In order to handle this, NoFS checks the inode status upon a `stat` call, and verifies the inode immediately if required, and then allows the system call to proceed. Since verification involves checking every data block referred to by the inode, the verification can take a lot of time. Running `ls -l` on a large directory of unverified files involves a large performance penalty arising from reading every file. For verified inodes, the `stat` will always return valid data, as the inode's attributes are updated whenever an error is encountered on block access. Note that NoFS does not check directory entries for correctness.

In the case of an unverified data block, no additional I/O is incurred during reads and partial writes since both involve reading the block off the disk anyway. However, in the case of a block overwrite, the block has to be read first to verify that it belongs to the inode before overwriting it. As a result, a write in ext2 is converted into a read-modify-write in NoFS, effectively cutting throughput in half. It should be noted that this happens only on the *first* overwrite of each unverified block. After the first overwrite, the block has been verified, and hence the backpointer no longer needs to be checked.

Thus it can be seen that accessing unverified objects involves a large performance hit. However, these costs are only incurred during the window between file-system mount and scan completion.

5 Evaluation

We now evaluate NoFS in two categories: reliability and performance. For reliability testing, we artificially prevent writes to certain sectors from reaching the disk, and then observe how NoFS handles the resulting inconsistency.

			ext2		NoFS	
System call	Blocks dropped	Error	Detected?	Action?	Detected?	Action?
mkdir	C^{inode}	P^{BD}, C^{OB}	$\times$	–	$\surd$	R, C^{EI}
mkdir	C^{dir}	C^{BD}	$\surd$	C^{ED}	$\surd$	C^{ED}
mkdir	P^{dir}	C^{OI}, C^{OB}	$\times$	–	$\surd$	R
mkdir	C^{inode}, C^{dir}	P^{BD}, C^{BD}	$\times$	–	$\surd$	C^{EI}
mkdir	C^{inode}, P^{dir}	C^{OB}	$\times$	–	$\surd$	R
mkdir	C^{dir}, P^{dir}	C^{OI}	$\times$	–	$\surd$	R
link	C^{inode}	C^{HL}	$\times$	–	$\surd$	C^{EN}
link	P^{dir}	C^{OI}	$\times$	–	$\surd$	R
unlink	C^{inode}	C^{HL}	$\times$	–	$\surd$	C^{EO}
unlink	O^{dir}	P^{BD}	$\times$	–	$\surd$	C^{EI}
rename	N^{dir}	O^{BD}	$\times$	–	$\surd$	C^{EI}
rename	O^{dir}	C^{OI}	$\times$	–	$\surd$	R
write	C^{data}	C^{GD}	$\times$	–	$\surd$	C^{EB}
write	C^{ind}	C^{GD}	$\times$	–	$\surd$	C^{EB}
write	C^{inode}, C^{data}	C^{OB}	$\times$	–	$\surd$	R
write	C^{inode}, C^{ind}	C^{OB}	$\times$	–	$\surd$	R
write	C^{data}, C^{ind}	C^{GD}	$\times$	–	$\surd$	C^{EB}
delete-create	O^{dir}	O^{BD}	$\times$	–	$\surd$	C^{EO}
truncate-write	O^{inode}	O^{TP}	$\times$	–	$\surd$	O^{EB}
unlink-link	O^{dir}	O^{BD}	$\times$	–	$\surd$	C^{EO}

General Key

C	Child	*inode*	File inode
P	Parent	*dir*	Directory block
O	Old file/parent	*data*	Data block
N	New file/parent	*ind*	Indirect block

Key for Error		Key for Action	
BD	Bad dir entry	*R*	Block/inode reclaimed on scan
OB	Orphan block	*EI*	Error on inode access
OI	Orphan inode	*ED*	Error on data access
HL	Wrong hard link count	*EN*	Error on access via new path
GD	Garbage data	*EO*	Error on access via old path
TP	2 inodes refer to 1 block	*EB*	Error on block access

Table 3: **Reliability testing.** *The table shows how NoFS reacts to various inconsistencies that occur due to updates not reaching the disk. The behavior of ext2 is also shown. NoFS detects all inconsistencies and reports an error, while ext2 lets most of the errors pass by undetected.*

For performance testing, we evaluate the performance of NoFS on a number of micro and macro-benchmarks. We compare the performance of NoFS to ext2, an orderless file system with no consistency, and ext3 (in ordered mode), a journaling file system with metadata consistency.

5.1 Reliability

We test whether NoFS can handle inconsistencies caused by a file-system crash. When a crash happens, any subset of updates involved in a file-system operation could be lost. We emulate different system-crash scenarios by artificially restricting blocks from reaching the disk, and restarting the file-system module. The restarted module will see the results of a partially completed update on disk.

We use a pseudo-device driver to prevent writes on target blocks and inodes from reaching the disk drive. We interpose the pseudo-device driver in-between the file system and the physical device driver, and all writes to the disk drive go via the pseudo-device driver. The file system and the device driver communicate through a list of sectors. In the file system, we calculate the on-disk sectors of target blocks and inodes and add them to the black list of sectors. All writes to these sectors are ignored by the device driver. Thus, we are able to target inodes and blocks in a fine grained manner.

Table 3 lists the behavior of ext2 and NoFS when 20 different inconsistencies are caused by dropping some of the blocks involved in each file-system operation. For example, consider the *mkdir* operation. It involves adding a directory entry to the parent directory, updating the new child inode, and creating a new directory block for the child inode. We do not consider updates to the access time of the parent inode. In the reliability test, we would drop writes to different combinations of these blocks, and observe the actions taken by the file system. For instance, if the write to the new child inode is dropped, it creates a bad directory entry in the parent directory, and orphans the directory block of the new child inode. We observe whether the file system detects this corrupt directory entry, and whether the orphan block is reclaimed. Both these actions are performed successfully in NoFS, whereas ext2 allows the user to access a garbage inode, and the block remains an orphan until the next file-system check.

The table entries which have two system calls denote the second system call happening after the first system call. These particular combinations were selected because they share a common resource. For example, truncate-write explores the case when a data block is deleted from a file and reassigned to another file. If the write to the truncated file inode fails, both files now point to the same data block, leading to an inconsistency. Similarly unlink-link and delete-create may share the same inode.

Some inconsistencies, like a corrupt directory block, are detected by ext2. Many other inconsistencies, such as reading garbage data, are not detected by ext2. All inconsistencies are detected by NoFS, and an error is returned to the user. When blocks and inodes are orphaned due to a crash, they are reclaimed by NoFS when the file system is scanned for allocation information upon reboot. Some of the inconsistencies could lead to potential security holes: for example, linking a sensitive file for temporary access, and removing the link later. If the directory block is not written to disk, the file could still be accessed, providing a way to read sensitive information. These security holes are detected upon access in NoFS, and any operation on them leads to an error.

5.2 Performance

To evaluate the performance of NoFS, we run a series of micro-benchmark and macro-benchmark workloads. We also observe the performance of NoFS at mount time, when the scan threads are still active. We show that NoFS has comparable performance to ext2 in most workloads, and that the performance of NoFS is reasonable when the scan threads are running in the background. We also measure the scan running time when the file system is populated with data, the rate at which NoFS scans data blocks to find free space, and the performance cost incurred when the stat system call is run on unverified inodes.

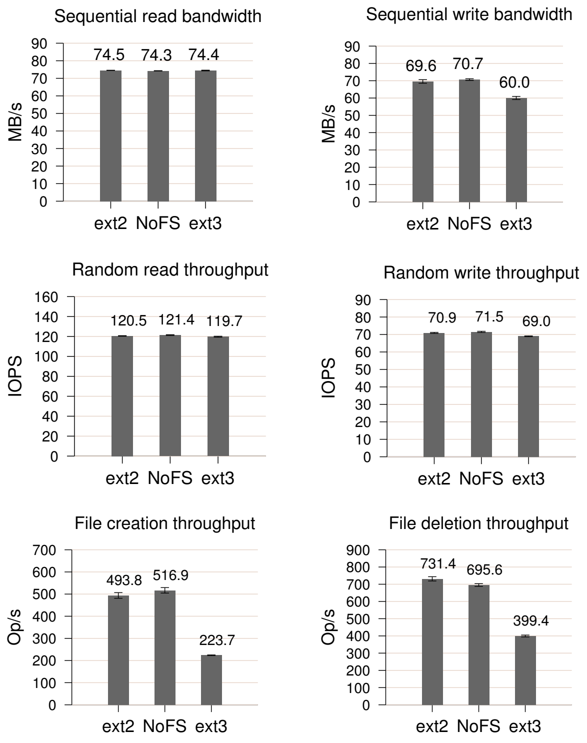


Figure 4: **Micro-benchmark performance.** *This figure compares file-system performance on various micro-benchmarks. The sequential benchmarks involve reading and writing a 1 GB file. The random benchmarks involve 10K random reads and writes in units of 4088 bytes (4096 bytes - 8 byte backpointer) across a 1 GB file, with a fsync after 1000 writes. The creation and deletion benchmarks involve 100K files spread over 100 directories, with a fsync after every create or delete.*

Our experiments were performed on a machine with a AMD 1 Ghz Opteron processor, and 1 GB of memory running Linux 2.6.27.55. The disk drive used in the experiment was a Seagate Barracuda 160 GB, which provides 75 MB/s read throughput and 70 MB/s write throughput. All experiments were performed on a cold file-system cache. The experiments were stable and repeatable. The numbers reported are the average over 10 runs.

5.2.1 Micro-benchmarks

We run a number of micro-benchmarks, focusing on different operations like sequential write and random read. Figure 4 illustrates the performance of NoFS on these workloads. We observe that NoFS has minimal overhead on the read and write workloads. For the sequential write workload, the performance of ext3 is worse than ext2 and NoFS due to the journal writes that ext3 performs.

The creation and deletion workloads involve doing a large number of creates/deletes of small files followed by fsync. This workload clearly brings out the performance penalty due to ordering points. The throughput of NoFS

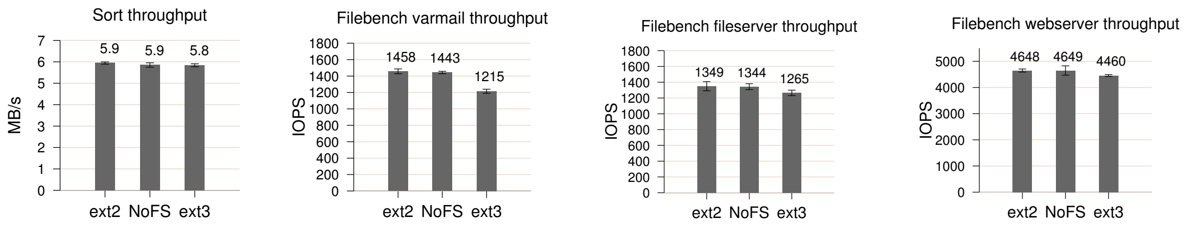


Figure 5: **Macro-benchmark performance.** *The figure shows the throughput achieved on various application workloads. The sort benchmark is run on 500 MB of data. The varmail benchmark was run with parameters 1000 files, 100K mean dir width, 16K mean file size, 16 threads, 16K I/O size and 16K mean append size. The file and webserver benchmarks were run with the parameters 1000 files, 20 dir width, 1 MB I/O size and 16K mean append size. The mean file size was 128K for the fileserver benchmark and 16K for the webserver benchmark. Fileserver benchmark used 50 threads while webserver used 100 threads.*

is twice that of ext3 on the file creation micro-benchmark, and 70% higher than ext3 on the file deletion benchmark.

5.2.2 Macro-benchmarks

We run the sort and Filebench [8] macro-benchmarks to assess the performance of NoFS on application workloads. Figure 5 illustrates the performance of the three file systems on this macro-benchmark. We selected the sort benchmark because it is CPU intensive. It sorts a 500 MB file generated by the *gensort* tool [22], using the command-line sort utility. The performance of NoFS is similar to that of ext2 and ext3, demonstrating that NoFS has minimal CPU overhead.

We run three workloads on Filebench: fileserver, webserver, and varmail. The fileserver workload emulates file-server activity, performing a sequence of creates, deletes, appends, reads, and writes. The webserver workload emulates a multi-threaded web host server, performing sequences of open-read-close on multiple files plus a log file append, with 100 threads. The varmail workload emulates a multi-threaded mail server, performing a sequence of create-append-sync, read-append-sync, reads, and deletes in a single directory.

We believe these benchmarks are representative of the different kind of I/O workloads performed on file systems. The performance of NoFS matches ext2 and ext3 on all three workloads. NoFS outperforms ext3 by 18% on the varmail benchmark, demonstrating the performance degradation in ext3 due to ordering points.

5.2.3 Scan performance

We evaluate the performance of NoFS at mount time, when the scanner is still scanning the disk for free resources. The scanner is configured to run every 60 seconds, and each run lasts approximately 16 seconds. In order to understand the performance impact due to scanning, we do two experiments involving 10 sequential writes of 200 MB each. The writes are spaced 30 seconds apart.

In the first experiment, we start the writes at mount time. The scanning of the disk and the sequential write is interleaved at 0s, 60s, 120s, and so on, leading to the write bandwidth dropping to half. When the sequential writes are run at 30s, 90s, 150s, and so on, the writes achieve peak bandwidth. In the second experiment, the writes were once again spaced 30s apart, but were started at 20s, after the end of the first scan run. In this experiment, the writes are never interleaved with the scan reads, and hence suffer no performance degradation. Graph (a) in Figure 6 illustrates these results.

Once the scan finishes, writes will once again achieve peak bandwidth. Running the scan runs without a break causes the scan to finish in around 90 seconds on an empty file system. Of course, one can configure this trade-off as need be; the larger the interval between scans, the smaller the performance impact during this phase, but the longer it takes to fully discover the free blocks of the system.

Graph (b) in Figure 6 depicts the time taken to finish the scan (both metadata and data) when the file system is increasingly populated with data. In this experiment, the scan is run without a break upon file-system mount. All the data in the file system are in units of 1 MB files. The running time of the scan increases slowly when the amount of data in the file system is increased, reaching about 140s for 1 GB of data. We also performed an experiment where we created a variable number of empty files in the file systems and measured the time for the scan to run. We found that the time taken to finish the scan remained the same irrespective of the number of empty files in the system. Since every inode in the system is read and verified, irrespective of whether it is actively used in the file system or not, the scan time remains constant.

During a file write, if there are no free blocks, the sequential block scanner is invoked in order to scan data blocks and find free space. The write will block until free space is found. Graph (c) illustrates the performance of the sequential block scanner. The latency to scan 100 MB is around 3 seconds, and 1 GB of data is scanned in around 30 seconds. The throughput is currently around 30 MB/s, so there is opportunity for optimizing its performance.

As mentioned in Section 4.4, when `stat` is run on an unverified inode, NoFS first verifies the inode by checking all its data blocks. We ran an experiment to estimate the cost of such verification. We created four identical directories, each filled with a number of 1 MB files. Every 140 seconds, `ls -li` was run on one directory, leading

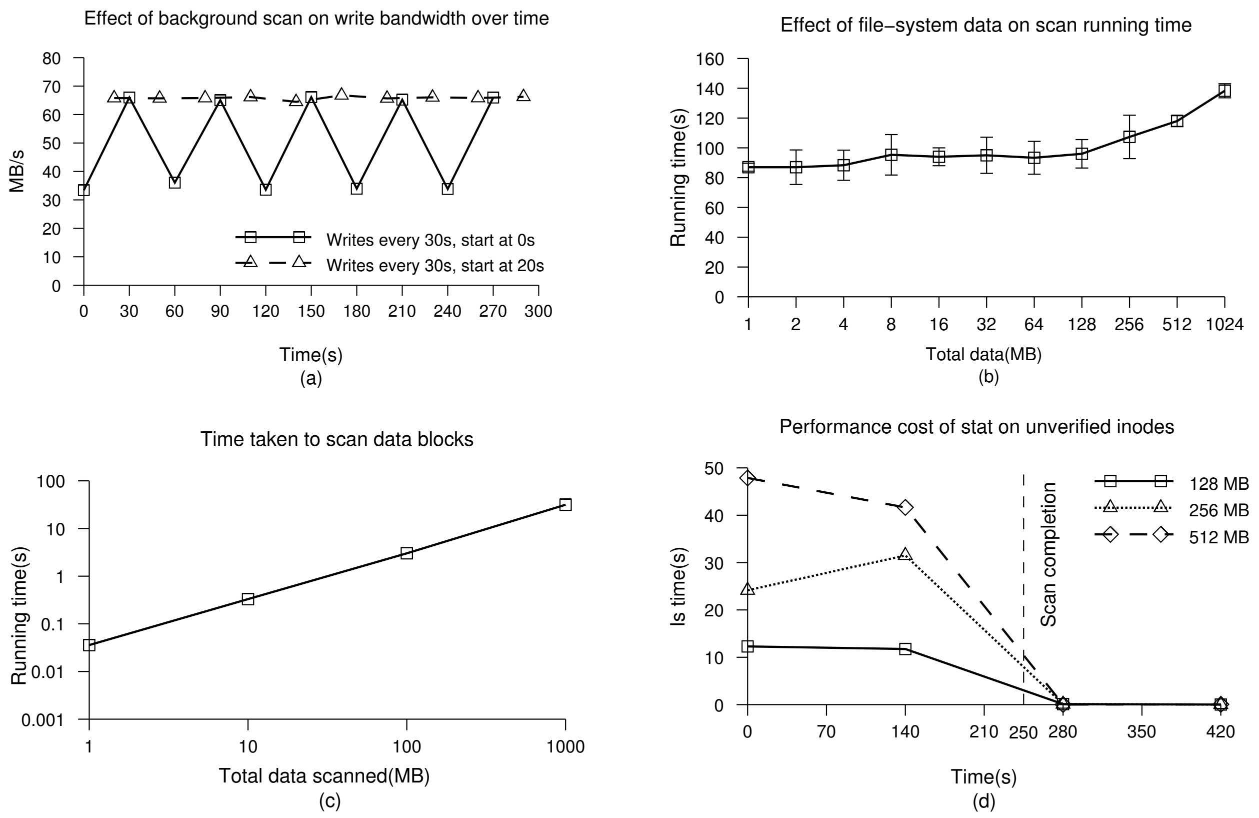


Figure 6: **Scan performance.** *Figure (a) depicts the reduction in write bandwidth when sequential writes interleave with the background scan. Figure (b) shows that the running of the scan increases slowly with the amount of data in the file system. Figure (c) illustrates the rate at which data blocks are scanned. Figure (d) demonstrates the performance cost incurred when the stat system call is run on unverified inodes.*

to a `stat` on each inode in the directory. The background scan started at file-system mount and finished at approximately 250 seconds. We varied the number of files from 128 to 512 and measured the time taken for `ls -li` in each experiment. Graph (d) illustrates the results. As expected, the time taken for `ls` to complete increases with the total data in the directory. After the scan completion at 250 seconds, all the inodes are verified, and hence `ls` finishes almost instantly.

6 Discussion

We have demonstrated that NoFS has better performance than journaling file systems such as ext3, while providing better consistency guarantees. However, it should be noted that NoFS differs from ext3 in two important aspects. First, it does not provide atomic transactions. Second, NoFS has no redundancy anywhere in the system. Part of the reason ext3 performs worse than NoFS is its extra log writes. By writing transaction updates to a log first, ext3 provides both metadata consistency, and the ability to preserve old data if the transaction fails before commit. NoFS only provides the former.

Given its current design, we feel an excellent use-case for NoFS would be as the local file system of a distributed file system such as the Google File System [11] or the Hadoop File System [37]. In such a distributed file system, reliable detection of corruption is all that is required, since redundant copies of data would be stored across the system. If the master controller is notified that a particular block has been corrupted in the local file system of a particular node, it can make additional copies of the data in order to counter the corruption of the block. Furthermore, such distributed file systems typically have large chunk sizes. As shown in section 5, NoFS provides very good performance on large sequential reads and writes, and is well suited for such workloads.

It should be noted that backpointer-based consistency could also be used to help ensure integrity in a conventional file system against bugs or data corruption. The simplicity and low overhead of backpointers makes such an addition to an existing file system feasible.

By eliminating ordering, backpointer-based consistency allows the file system to maintain consistency without depending upon lower-layer primitives such as the disk cache flush. Previous research has shown that SATA drives do not always obey the flush command [29, 36], which is essential for file systems to implement ordering. IDE drives have also been known to disobey flush commands [28, 39]. Using backpointer-based consistency allows a file system to run on top of such misbehaving disks and yet maintain consistency.

Potential users of NoFS should note two things. One, any application which requires strict ordering among file creates and writes should not use NoFS. Two, if there are corrupt files in the system, NoFS will only detect them upon access and not upon file-system mount. Some users may prefer to find out about corruption at mount time rather than when the file system is running. Such a use case aligns better with a file system such as ext3.

7 Related Work

The idea of using information inside or near the block to detect errors is not new. Cambridge File Server [7] used certain bits in each cylinder (cylinder map) to store the allocation status of blocks in that cylinder. Cedar File System [12] used 'labels' inside pages to check their allocation status. Embedding logical identity of blocks (inode number + offset) has been done in RAID to recover from lost and misdirected writes [16]. Transactional flash [27] embeds commit records inside every page to provide transactions and recovery. However, NoFS is the first work that we know of that clearly defines the level of consistency that such information provides and uses such information alone to provide consistency.

The design of the Pilot file system [30] is very similar to that of NoFS. Pilot employs self identifying pages and uses a scavenger to reconstruct the file system metadata upon crash. However, like the file-system check, the scavenger needs to finish running before the file system can be accessed. In NoFS, the file system is made available upon mount, and can be accessed while the scan is running in the background.

Pangaea [33] uses backpointers for consistency in a distributed wide area file system. However, its use of backpointers is limited to directory entry backpointers that are used to resolve conflicting updates on directories. Similar to NoFS, Pangaea also uses the backpointer as the true source of information, letting the backpointers of child inodes dictate whether they belong to a directory or not.

btrfs [48] supports back references that allow it to obtain the list of the extents that refer to a particular extent. Although back references are conceptually similar to NoFS backpointers, the main purpose of btrfs back references is supporting efficient data migration, rather than providing consistency. Other mechanisms such as checksums are used to ensure that the data is not corrupt in btrfs. Another key difference is that btrfs does not always store the back reference inside the allocated extent: sometimes the back references are stored as separate items close to the extent allocation records.

Backlog [17] also uses explicit back references in order to manage migration of data in write anywhere file systems. The back references in Backlog are stored in a separate database, and are designed for efficient querying of usage information rather than consistency. Backlog's back references are not used for incremental file-system checking or resolving ownership disputes.

While NoFS makes an order-less file system more available by eliminating the need for the file-system check, there have been other approaches to increasing availability such as doing the file-system check while the system is online. McKusick's background fsck [19] could repair simple inconsistencies such as lost resources by running fsck on snapshots of a running system. Chunkfs [14] is similar to our work, providing incremental, online file-system checking. Chunkfs differs from NoFS in that the minimal unit of checking is a chunk whereas it is a single file or block in NoFS. Chunkfs does not offer online repair of the file system, while it is possible in NoFS, due to backpointers and non-persistent allocation structures.

8 Conclusion

Every modern file system uses ordering points to ensure consistency. However, ordering points have many disadvantages including lower performance, higher complexity in file-system code, and dependence on lower layers of the storage stack to enforce ordering of writes.

In this paper, we demonstrate that it is possible to build an order-less file system, NoFS, that provides consistency without sacrificing simplicity, availability or performance. NoFS allows immediate data access upon mounting, without file-system checks. We show that NoFS has excellent performance on many workloads, outperforming ext3 on workloads that frequently flush data to disk explicitly.

Although potentially useful for the desktop, we believe NoFS may be of special significance in cloud computing platforms, where many virtual machines are multiplexed onto a physical device. In such cases, the underlying host operating system may try to batch writes together for performance, potentially ignoring ordering requests from virtual machines. NoFS allows virtual machines to maintain consistency without depending on the numerous lower layers of software and hardware. Removing such trust is key to building more robust and reliable storage systems.

Acknowledgments

We thank Benjamin Reed (our shepherd), S. Subramanian, S. Sundaraman, I. Ahuja, L. Lu, Y. Zhang, M. Saxena, S. Panneerselvam, A. Kumar, T. Harter, A. Anand, T. Do, and the anonymous reviewers for their feedback. This material is based upon work supported by the NSF under CCF-0811657 and CNS-0834392 as well as donations from Google, NetApp, and Samsung. Any opinions, findings, and conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF or other institutions.

References

[1] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.pdf.

[2] Steve Best. JFS Overview. `www.ibm.com/developerworks/library/l-jfs.html`, 2000.

[3] Jeff Bonwick and Bill Moore. ZFS: The Last Word in File Systems. `http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf`, 2007.

[4] Remy Card, Theodore Ts'o, and Stephen Tweedie. Design and Implementation of the Second Extended Filesystem. In Proceedings of the First Dutch International Symposium on Linux, 1994.

[5] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Consistency Without Ordering. Technical Report 1709, University of Wisconsin-Madison Computer Sciences, January 2012.

[6] Charles D. Cranor and Gurudatta M. Parulkar. The UVM Virtual Memory System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '99)*, Monterey, California, June 1999.

[7] Jeremy Dion. The Cambridge File Server. *SIGOPS Operating Systems Review*, 14:26–35, October 1980.

[8] Stony Brook University File system Storage Lab (FSL). Filebench Benchmark. `http://sourceforge.net/apps/mediawiki/filebench/index.php?title=Filebench`, 2011.

[9] Christopher Frost, Mike Mammarella, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoka, and Lei Zhang. Generalized File System Dependencies. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, Stevenson, Washington, October 2007.

[10] Gregory R. Ganger and Yale N. Patt. Metadata Update Performance in File Systems. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI '94)*, pages 49–60, Monterey, California, November 1994.

[11] 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.

[12] Robert Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP '87)*, Austin, Texas, November 1987.

[13] Val Henson, Zach Brown, Theodore Ts'o, and Arjan van de Ven. Reducing Fsck Time For Ext2 File Systems. In *Ottawa Linux Symposium (OLS '06)*, Ottawa, Canada, July 2006.

[14] Val Henson, Arjan van de Ven, Amit Gud, and Zach Brown. Chunkfs: Using Divide-And-Conquer to Improve File System Reliability and Repair. In *IEEE 2nd Workshop on Hot Topics in System Dependability (HotDep '06)*, Seattle, Washington, November 2006.

[15] 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.

[16] Andrew Krioukov, Lakshmi N. Bairavasundaram, Garth R. Goodson, Kiran Srinivasan, Randy Thelen, Andrea C. Arpaci-Dusseau, and Remzi 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.

[17] Peter Macko, Margo Seltzer, and Keith A. Smith. Tracking Back References in a Write-Anywhere File System. In *Proceedings of the 8th USENIX conference on File and storage technologies*, San Jose, California, February 2010.

[18] Marshall K. McKusick, William N. Joy, Sam J. Leffler, and Robert S. Fabry. A Fast File System for

UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[19] Marshall Kirk McKusick. Running 'fsck' in the Background. In *Proceedings of BSDCon 2002 (BSDCon '02)*, San Fransisco, CA, February 2002.

[20] Marshall Kirk McKusick, Willian N. Joy, Samuel J. Leffler, and Robert S. Fabry. Fsck - The UNIX File System Check Program. Unix System Manager's Manual - 4.3 BSD Virtual VAX-11 Version, April 1986.

[21] Rajeev Nagar. *Windows NT File System Internals: A Developer's Guide*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1997.

[22] Chris Nyberg. gensort Data Generator. `http://www.ordinal.com/gensort.html`, 2009.

[23] David Patterson, Garth Gibson, and Randy Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD Conference on the Management of Data (SIGMOD '88)*, pages 109–116, Chicago, Illinois, June 1988.

[24] J. Kent Peacock, Ashvin Kamaraju, and Sanjay Agrawal. Fast Consistency Checking for the Solaris File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '98)*, pages 77–89, New Orleans, Louisiana, June 1998.

[25] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '05)*, pages 105–120, Anaheim, California, April 2005.

[26] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi 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, October 2005.

[27] Vijayan Prabhakaran, Thomas L. Rodeheffer, and Lidong Zhou. Transactional Flash. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, California, December 2008.

[28] R1Soft. Disk Safe Best Practices. `http://wiki.r1soft.com/display/CDP3/Disk+Safe+Best+Practices`, December 2011.

[29] Abhishek Rajimwale, Vijay Chidambaram, Deepak Ramamurthi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Coerced Cache Eviction and Discreet-Mode Journaling: Dealing with Misbehaving Disks. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'11)*, Hong Kong, China, June 2011.

[30] David D. Redell, Yogen K. Dalal, Thomas R. Horsley, Hugh C. Lauer, William C. Lynch, Paul R. McJones, Hal G. Murray, and Stephen C.Purcell. Pilot: An Operating System for a Personal Computer. *Communications of the ACM*, 23(2):81–92, February 1980.

[31] Hans Reiser. ReiserFS. `www.namesys.com`, 2004.

[32] 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.

[33] Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam. Taming Aggressive Replication in the Pangaea Wide-Area File System. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[34] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-To-End Arguments in System Design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[35] Eric Schrock. UFS/SVM vs. ZFS: Code Complexity. `http://blogs.sun.com/eschrock/`, November 2005.

[36] Seagate Forums. ST3250823AS (7200.8) ignores FLUSH CACHE in AHCI mode. `http://bit.ly/xcSAUV`, September 2011.

[37] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *MSST '10*, Incline Village, Nevada, May.

[38] Muthian Sivathanu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Somesh Jha. A Logic of File Systems. In *Proceedings of the 4th USENIX Symposium on File and Storage Technologies (FAST '05)*, pages 1–15, San Francisco, California, December 2005.

[39] SQLite. How To Corrupt Your Database Files. `http://www.sqlite.org/lockingv3.html`.

[40] Sun Microsystems. ZFS: The last word in file systems. `www.sun.com/2004-0914/feature/`, 2006.

[41] 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.

[42] Technical Committee T10. T10 Data Integrity Field standard. `http://www.t10.org/`, 2009.

[43] The Serial ATA International Organization. Serial ATA Revision 3.0 Specification. `http://www.sata-io.org/technology/6Gbdetails.asp`, June 2009.

[44] Theodore Ts'o. `http://e2fsprogs.sourceforge.net`, June 2001.

[45] Stephen C. Tweedie. Journaling the Linux ext2fs File System. In *The Fourth Annual Linux Expo*, Durham, North Carolina, May 1998.

[46] Stephen C. Tweedie. EXT3, Journaling File System. `olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html`, July 2000.

[47] VirtualBox Manual. Responding to guest IDE/SATA flush requests. `http://www.virtualbox.org/manual/ch12.html`.

[48] Wikipedia. Btrfs. `en.wikipedia.org/wiki/Btrfs`, 2009.

[49] Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, Washington, November 2006.

[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] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. End-to-end Data Integrity for File Systems: A ZFS Case Study. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST '10)*, San Jose, California, February 2010.

Reducing SSD Read Latency via NAND Flash Program and Erase Suspension

Guanying Wu and Xubin He
Department of Electrical and Computer Engineering
Virginia Commonwealth University, Richmond, VA 23284

Abstract

In NAND flash memory, once a page program or block erase (P/E) command is issued to a NAND flash chip, the subsequent read requests have to wait until the time-consuming P/E operation to complete. Preliminary results show that the lengthy P/E operations may increase the read latency by 2x on average. As NAND flash-based SSDs enter the enterprise server storage, this increased read latency caused by the contention may significantly degrade the overall system performance. Inspired by the internal mechanism of NAND flash P/E algorithms, we propose in this paper a low-overhead P/E suspension scheme, which suspends the on-going P/E to service pending reads and resumes the suspended P/E afterwards. In our experiments, we simulate a realistic SSD model that adopts multi-chip/channel and evaluate both SLC and MLC NAND flash as storage materials of diverse performance. Our experimental results show that the proposed technique achieves a near-optimal performance gain on servicing read requests. Specifically, the read latency is reduced on average by 50.5% compared to RPS and 75.4% compared to FIFO at cost of less than 4% overhead on write requests.

1 Introduction

NAND flash-based SSDs have better random access performance over hard drives and have potential in high performance computing system market. However, NAND flash has performance and cost problems which limit its application [11]. The problem addressed in this paper is the read vs. program/erase (P/E) contention. Due to slow P/E speed of NAND flash, once P/E is committed to the flash chip, pending or subsequent read requests suffer from the prolonged service latency caused by the waiting time. As disk read requests are resulted from upper level cache misses, the compromised read latency of the disk causes degraded application performance. To reduce read latency, on-disk write buffers may avoid or postpone the write commitments to the flash [9, 6, 7]. Executing the garbage collection processes during the idle time of the drive may also alleviate the contention between read and P/E [1, 10]. Furthermore, the read requests can be prioritized in a pending list to reduce the queuing time caused by the P/E. However, none of these approaches preempt the committed P/E for read requests.

To address this read vs. P/E contention problem, we propose a *P/E Suspension* scheme for NAND flash that allows the execution of the P/E operations to be suspended so as to service the pending reads and then the suspended P/E is resumed. The internal process of the program operation is done in a "step-by-step" fashion (Incremental Step Pulse Programming, or ISPP [2]), and thus the program can be suspended at the interval of two consecutive steps, or the on-going step could be canceled and re-executed upon resumption. The erase process requires the duration of erase-voltage pulse to be satisfied, and thus the erase can also be suspended and resumed as long as we ensure the required timing.

The implementation of P/E suspension for NAND flash involves minimal modifications to the flash interface, i.e., merely the *"program suspend/resume"* and *"erase suspend/resume"* commands need to be added in the command set of the flash interface [12]. To support P/E suspension, the *control logic inside the flash chip* is required to determine the appropriate time to suspend the P/E (suspension point) and to maintain or retrieve the previous state of the suspended P/E so as to resume it. Noting that the implementation feasibility of the proposed schemes is based on the fundamental/typical circuitry of flash memories [3].

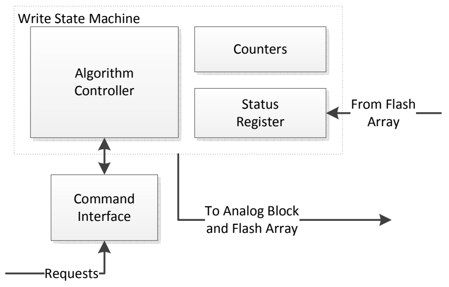
This paper makes the following contributions. First, we analyze the impact of the long P/E latency on read performance, showing that even with the read prioritization scheduling, the read latency is still severely compromised. Second, by exploiting the internal mechanism of the P/E algorithms in NAND flash memory, we propose a low-overhead P/E suspension scheme which suspends the on-going P/E operations for servicing the pending read requests. In particular, two strategies for suspending the program operation, *Inter Phase Suspension (IPS)* and *Intra Phase Cancelation(IPC)* are proposed. Third, based on simulation experiments under various workloads, we demonstrate that compared to FIFO, the proposed design can significantly reduce the SSD read latency for both SLC and MLC NAND flash.

The rest of this paper is organized as follows: In Section 2 we give an overview of the internal mechanism for P/E on NAND flash and briefly discuss related work. In Section 3, we conduct simulations to show how the read latency is increased by chip contention. We describe our detailed P/E suspension scheme in Section 4 and evaluate our approach via simulation experiments in Section 5. Finally we conclude our paper in Section 6.

2 Background and Related Work

2.1 NAND Flash Program/Erase Algorithm

Incremental Step Pulse Programming (ISPP) is typically used for precisely programming or erasing the NAND flash [3]. It is made of a series of program and verify iterations. The execution of ISPP and the erase process is implemented in the flash chip with an analog block and a control logic block. The analog block is responsible for regulating and pumping the voltage for program or erase operations. The control logic block is responsible for interpreting the interface commands, generating the control signals for the flash cell array and the analog block, and executing the program and erase algorithms. As shown in the following diagram [3], the write state machine consists of three components: an *algorithm controller* to execute the algorithms for the two types of operations, several *counters* to keep track of the number of ISPP iterations, and a *status register* to record the results from the verify operation.



2.2 Related Work

The idea of preempting low priority operations for high priority ones via breaking down an operation to small phases has been embodied in [4], [13], etc. Dimitrijevic et al. proposed *Semi-preemptible IO* [4] to divide HDD I/O requests to small disk commands to enable preemption for high priority requests. Similar to NAND flash, *Phase Change Memory* (PCM) has much larger write latency than read latency. Qureshi *et al.* proposed in [13] a few techniques to preempt the on-going writes of PCM for reads: *write cancelation* and a threshold-based overhead control method to reduce the overhead are proposed to cancel entire write operations; PCM, like NAND flash, adopts the *iterative-write* algorithm. Our work differs from [13] as follows: PCM has the in-place update capability, while NAND flash requires erase before program. In our work, the suspension of erase operation is proposed. *Write Cancelation* for the entire write process of NAND flash is not viable. NAND flash's iterative write process differs from PCM in that, each iteration has two phases (program and verify). Thus, for each iteration, we may have two suspension points. Furthermore, we propose the *shadow buffer* to overcome the overhead of re-transferring the write data upon resumption, which is not discussed in [13].

3 Motivation

In this section, we demonstrate how the read vs. P/E contention increases the read latency under various workloads. We have modified *MS-add-on* simulator [1] based on Disksim 4.0. Specifically, under the workloads of a variety of popular disk traces, we compare the read latency of two scheduling policies, FIFO and read priority scheduling (RPS), to show the limitation of RPS. Furthermore, with RPS, we set the latency of program and erase operation to be equal to that of read and *zero* to justify the impact of P/E on the read latency.

3.1 Configurations and Workloads

The simulated SSD is configured as follows: there are 16 flash chips, each of which owns a dedicated channel to the flash controller. Each chip has four planes that are organized in a RAID-0 fashion; the size of one plane is 512 MB or 1 GB assuming the flash is used as SLC or 2-bit MLC, respectively (the page size is 2 KB for SLC or 4 KB for MLC). To maximize the concurrency, each individual plane has its own allocation pool [1]. The garbage collection processes are executed in the background so as to minimize the interference with the foreground requests. In addition, the percentage of flash space over-provisioning is set as 30%, which doubles the value suggested in [1]. Considering the limited working-set size of the workloads used in this paper (described in next subsection), 30% over-provisioning is believed to be sufficient to avoid frequent execution of garbage collection processes. The write buffer size is 64 MB. The SSD is connected to the host via a PCI-E of 2.0 GB/s. The physical operating parameters of the flash memory is summarized in Table 1.

We choose 6 disk I/O traces for our experiments: *Financial 1 and 2* (F1, F2) [14]; *Display Ads Platform and payload servers* (DAP) and *MSN storage metadata* (MSN) traces [8]; *Cello99* [5] traces (C3, C8). Noting that those traces were originally collected on HDDs, to produce more stressful workloads for SSDs, we compress all these traces so that the system idle time is reduced from 98% to around 70% for each workload.

Symbols	Description	Value	
		SLC	MLC
T_{bus}	The bus latency of transferring one page from/to the chip	20 μs	40 μs
T_{r_phy}	The latency of sensing/reading data from the flash	10 μs	25 μs
T_{w_total}	The total latency of ISPP in flash page program	140 μs	660 μs
N_{w_cycle}	The number of ISPP iterations	5	15
T_{w_cycle}	The time of one ISPP iteration (T_{w_total}/N_{w_cycle})	28 μs	44 μs
$T_{w\_program}$	The duration of program phase of one ISPP iteration	20 μs	20 μs
T_{verify}	The duration of the verify phase	8 μs	24 μs
T_{erase}	The duration of erase pulse	1.5 ms	3.3 ms
$T_{voltage\_reset}$	The time to reset operating voltages of on-going operations	4 μs	
T_{buffer}	The time taken to load the page buffer with data	3 μs	

Table 1: Flash Parameters

Trace	SLC		MLC	
	Read	Write	Read	Write
F1	0.37	0.87	0.44	1.58
F2	0.24	0.57	0.27	1.03
DAP	1.92	6.85	5.74	11.74
MSN	4.13	4.58	8.47	25.21
C3	0.25	2.85	0.52	6.30
C8	0.44	2.33	0.56	4.54

Table 2: Numerical Latency Values of FIFO (in *ms*)

3.2 Experimental Results

In this subsection, we compare the read latency performance under four scenarios: FIFO; RPS; PER (the latency of program and erase is set equal to that of read); and PE0 (the latency of program and erase is set to zero). Note that both PER and PE0 are applied upon RPS in order to study the chip contention and the limitation of RPS. Due to the large range of the numerical values of the experimental results, we normalize them to the corresponding results of FIFO, which are listed in Table 2 for reference. The normalized results are plotted in Fig. 1, where the left part shows the results of SLC and the right part is for MLC. Compared to FIFO, RPS achieves impressive performance gain, e.g., the gain maximizes at an effective read latency ("effective" refers to the actual latency taking the queuing delay into account) reduction of 44.6% (SLC) and 48.3% (MLC) on average. However, if the latency of P/E is the same as read latency or zero, i.e., in the case of *PER* and *PE0*, the effective read latency can be further reduced. For example, with PE0, the read latency reduction is 71.7% (SLC) and 75.6% (MLC) on average. Thus, even with RPS policy, the chip contention still increases the read latency by about 2x on average.

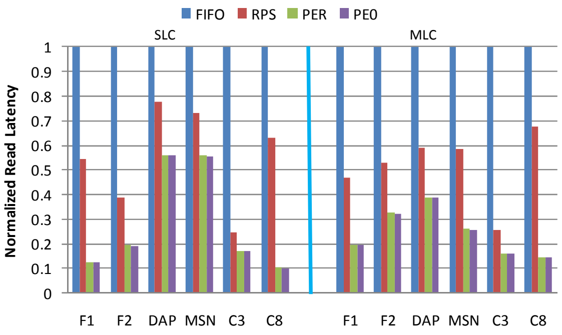


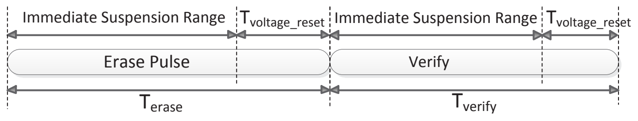
Figure 1: Read Latency Performance Comparison: FIFO, RPS, PER, and PE0. Results normalized to FIFO.

4 Design of P/E Suspension Scheme

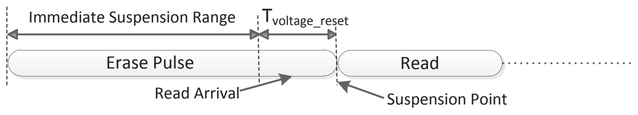
4.1 Erase Suspension and Resumption

In NAND flash, the erase process consists of two phases: first, an erase pulse lasting for T_{erase} is applied on the target block; second, a verify operation that takes T_{verify} is performed to check if the preceding erase pulse has successfully erased all bits in the block. Otherwise, the above process is repeated until success, or if the number of iterations reaches the predefined limit, an operation failure is reported. Typically, for NAND flash, since the *over-erasure* is not a concern [3], the erase operation can be done with a single erase pulse.

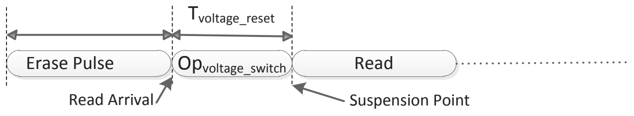
How to suspend an erase operation: suspending either the erase pulse or verify operation requires resetting the status of the corresponding wires that connect the flash cells with the analog block. Specifically, due to the fact that the flash memory works at different voltage bias for different operations, the current voltage bias applied on the wires (and thus on the cell) needs to be reset for the pending read request. This process ($Op_{voltage\_reset}$ for short) takes a period of $T_{voltage\_reset}$. Noting that either the erase pulse or verify operation always has to conduct $Op_{voltage\_reset}$ at the end (as shown in the following diagram of erase operation timeline).



Thus, if the suspension command arrives during $Op_{voltage\_reset}$, the suspension will succeed once $Op_{voltage\_reset}$ is finished (as illustrated in the following diagram of erase suspension timeline).



Otherwise, an $Op_{voltage\_reset}$ is executed immediately and then the read request is serviced by the chip (as illustrated in the following diagram).

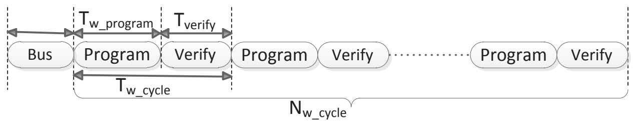


How to resume an erase operation: the resumption means the control logic of NAND flash resumes the suspended erase operation. Therefore, the control logic should keep track of the progress, i.e., whether the suspension happens during the verify phase or the erase pulse. For the first scenario, the verify operation has to be re-done all over again. For the second scenario, the erase pulse time left (T_{erase} minus the progress), for example, 1 ms will be done in the resumption if no more suspension happens. Actually, the task of progress tracking can be easily supported by the existing facilities in the control logic of NAND flash: the pulse width generator is implemented using a counter-like logic [3], which keeps track of the progress of the current pulse.

The overhead on the effective erase latency: resuming the erase pulse requires extra time to set the wires to the corresponding voltage bias, which takes approximately the same amount of time as $T_{voltage\_reset}$. Suspending during the verify phase causes a re-do in the resumption, and thus the overhead is the time of the suspended/cancelled verify operation. In addition, the read service time is included in the effective erase latency.

4.2 Program Suspension and Resumption

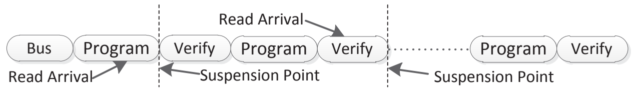
The process of servicing a program request is: first, the data to be written is transferred through the controller-chip bus and loaded in the page buffer; then the ISPP is executed, in which a total number of N_{w_cycle} iterations consisting of a *program* phase followed by a *verify* phase are conducted on the target flash page. In each ISPP iteration, the program phase is responsible for applying the required program voltage bias on the cells so as to charge them. In the verify phase, the content of the cells is read to verify if the desired amount of charge is stored in each cell: if so, the cell is considered *program-completion*; otherwise, one more ISPP iteration will be conducted on the cell. Due to the fact that all cells in the target flash page are programmed simultaneously, the overall time taken to program the page is actually determined by the cell that needs the most number of ISPP iterations. A major factor that determines the number of ISPP iterations needed is the amount of charge to be stored in the cell, which is in turn determined by the data to be written. For example, for the 2-bit MLC flash, programming a “0” in a cell needs the most number of ISPP iterations, while for “3” (the erased state), no ISPP iteration is needed. Since all flash cells in the page are programmed simultaneously, N_{w_cycle} is determined by the smallest data (2-bit) to be written; nonetheless, we make a rational assumption in our simulation experiments that N_{w_cycle} is constant and equal to the maximum value. The program process is illustrated in the following diagram.



How to retain the page buffer content: before we move on to suspension, this critical problem has to be solved. For program, the page buffer contains the data to be written. For read, it contains the retrieved data to be transferred to the flash controller. If a write is preempted by a read, the content of the page buffer is certainly replaced. Thus, the resumption of the write demands the page buffer re-stored. Intuitively, the flash controller that is responsible for issuing the suspension and resumption commands may keep a copy of the write page data until the program is finished and upon resumption, the controller re-sends the data to the chip through the controller-chip bus. However, the page transfer consumes a significant amount of time: unlike the NOR flash which does *byte programming*, NAND flash does *page programming*, and the page size is of a few kilobytes. For instance, assuming a 100 MHz bus and 4 KB page size, the bus time T_{bus} is about 40 μs.

To overcome this overhead, we propose a *Shadow Buffer* in the flash. The shadow buffer serves like a replica of the page buffer and it automatically loads itself with the content of the page buffer upon the arrival of the write request and re-stores the page buffer while resumption. The load and store operation takes the time T_{buffer}. The shadow buffer has parallel connection with the page buffer, and thus the data transfer between them can be done on the fly. T_{buffer} is normally smaller than T_{bus} by one order of magnitude.

How to suspend a program operation: compared to the long width of the erase pulse (T_{erase}), the program and verify phase of the program process is normally two orders of magnitude shorter. Intuitively, the program process can be suspended at the end of the program phase of any ISPP iteration as well as the end of the verify phase. We refer to this strategy as “Inter Phase Suspension” (**IPS**). IPS has in total $N_{w\_cycle} * 2$ potential suspension points as illustrated in the following diagram.



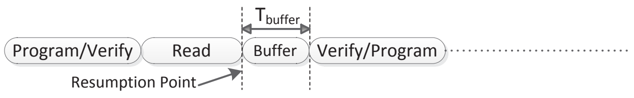
Due to the fact that at the end of the program or verify phase, the status of the wires has already reset ($Op_{voltage\_reset}$), IPS does not introduce any extra overhead, except for the service time of the read or reads that preempt the program. However, the effective read latency should include the time from the arrival of read to the end of the corresponding phase. For simplicity, assuming the arrival time of reads follows the uniform distribution, the probability of encountering the program phase and the verify phase is $T_{w\_program}/(T_{verify}+T_{w\_program})$ and $T_{verify}/(T_{verify}+T_{w\_program})$, respectively. Thus, the average extra latency for the read can be calculated as:

$$T_{read\_extra} = \frac{T_{w\_program}}{(T_{verify}+T_{w\_program})} * \frac{T_{w\_program}}{2} + \frac{T_{verify}}{(T_{verify}+T_{w\_program})} * \frac{T_{verify}}{2} \qquad (1)$$

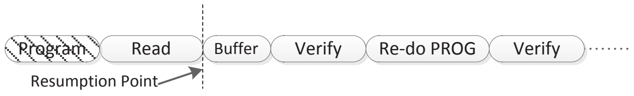
Substituting the numerical values in Table 1, we get 8.29 μs (SLC) and 11.09 μs (MLC) for T_{read_extra}, which is comparable to the physical access time of the read (T_{r_phy}). To further improve the effective read latency, we propose "Intra Phase Cancelation" (**IPC**). Similar to canceling the verify phase for the erase suspension, IPC cancels an on-going program or verify phase upon suspension. The reason of canceling instead of pausing the program phase is that the duration of the program phase, $T_{w\_program}$, is short and normally considered atomic (cancelable but not pause-able).

Again, for IPC, if the read arrives when the program or verify phase is conducting $Op_{voltage\_reset}$, the suspension happens actually at the end of the phase, which is the same as IPS; otherwise, $Op_{voltage\_reset}$ is started immediately and the read is then serviced. Thus, IPC achieves a T_{read_extra} no larger than $T_{voltage\_reset}$.

How to resume from IPS: first of all, the page buffer is re-loaded with the content of the shadow buffer. Then, the control logic examines the last ISPP iteration number and the previous phase. If IPS happens at the end of the verify phase, which implies that the information of the status of cells has already been obtained, we may continue with the next ISPP if needed; on the other hand, if the last phase is the program phase, naturally we need to finish the verify operation before moving on to the next ISPP iteration. The resumption process is illustrated in the following diagram.



How to resume from IPC: compared to IPS, the resumption from IPC is more complex. Different from the verify operation, which does not change the charge status of the cell, the program operation puts charge in the cell and thus changes the threshold voltage (V_{th}) of the cell. Therefore, we need to determine whether the canceled program phase has already achieved the desired V_{th} (i.e., whether the data could be considered written in the cell), by a verify operation. If so, no more ISPP iteration is needed on this cell; otherwise, the previous program operation is executed on the cell again. The later case is illustrated in the following diagram.



Re-doing the program operation would have some affect on the tightness of V_{th}, but with the aid of ECC and a fine-grained ISPP, i.e., small incremental voltage ΔV_{pp}, the IPC has little impact on the data reliability of the NAND flash. The relationship between ΔV_{pp} and the tightness of V_{th} is modeled in [15].

The overhead on the effective write latency: IPS requires re-loading the page buffer, which takes T_{buffer}. For IPC, if the verify phase is canceled, the overhead is the time elapsed of the canceled verify phase plus the read service time and T_{buffer}. In case of program phase, there are two scenarios: if the verify operation reports that the desired V_{th} is achieved, the overhead is the read service time plus T_{buffer}; otherwise, the overhead is the time elapsed of the canceled program phase plus an extra verify phase, in addition to the overhead of the above scenario. Clearly, IPS achieves smaller overhead on the write than IPC but relatively lower read performance.

5 Performance Evaluation

In this section, we evaluate our proposed design under different workloads described in Section 3.1.

5.1 Read Performance Gain

First, we compare the average read latency of P/E suspension with RPS, PER and PE0 in Fig. 2, where the results are normalized to that of RPS. For P/E suspension, the IPC (Intra Phase Cancelation), denoted as "PES_IPC", is adopted in Fig. 2. PE0, with which the physical latency values of program and erase are set to zero, serves as an optimistic situation where the contention between reads and P/E's is completely eliminated. Fig. 2 demonstrates that, compared to RPS, the proposed P/E suspension achieves a significant read performance gain, which is almost equivalent to the optimal case, PE0 (with less than 1% difference). Specifically, on the average of the 6 traces, PES_IPC reduces the read latency by 48.9% for SLC and 50.5% for MLC compared to RPS, and 71.6% for SLC and 75.4% for MLC compared to FIFO. For conciseness, the results of SLC and (then) MLC are listed without explicit specification in the following text.

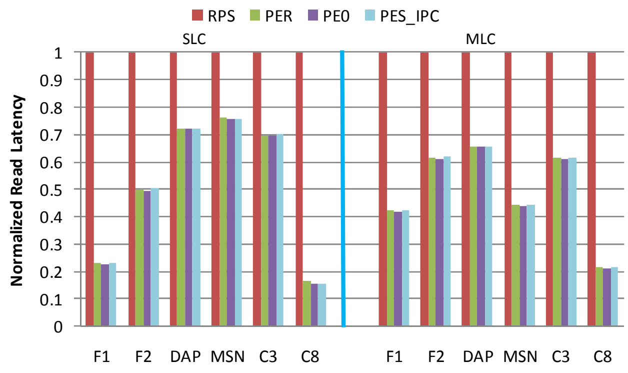


Figure 2: Read Latency Performance Comparison: RPS, PER, PE0, and PES_IPC (P/E Suspension using IPC). Normalized to RPS.

As stated in Section 4, IPC can achieve better read performance but cause higher write overhead compared to IPS. We compare the read performance of IPC and IPS in Fig. 3. The read latency of IPS is 8.0% and 2.7% on average and at-most 13.2% and 6.7% (under F1) higher than that of IPC. The difference is resulted from the fact that IPS has extra read latency, which is mostly the time between read request arrivals and the suspension points at the end of the program or verify phase. We notice that the latency performance of IPS using SLC is poorer than MLC under all traces, which is because of the higher sensitivity of SLC's read latency to the overhead caused by the extra latency.

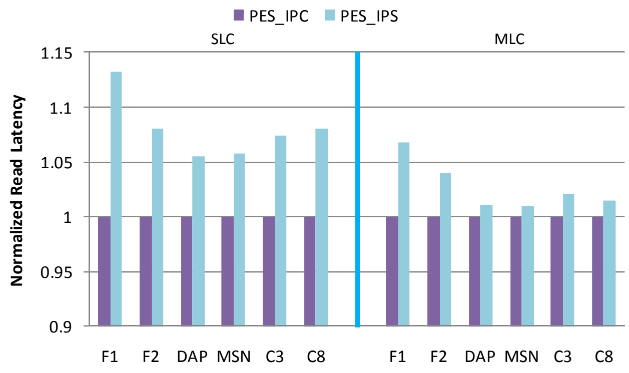


Figure 3: Read Latency Performance Comparison: PES_IPC vs. PES_IPS. Normalized to PES_IPC.

5.2 Write Overhead

Since both RPS and P/E suspension introduce minimal extra chip bandwidth usage, the write throughput is barely compromised. We use the latency as a metric for the overhead evaluation. First, we compare the average write latency of FIFO, RPS, PES_IPS, and PES_IPC in Fig. 4. Obviously, the write overhead in terms of latency is trivial compared to the read performance gain we achieve with P/E suspension. Specifically, RPS increases the write latency by 2.3% and 1.2% on average and at-most 6.7% (SLC, MSN) and 3.8% (MLC, DAP), compared to FIFO. PES_IPC increases write latency by 3.6% and 1.9% on average and at-most 6.9% (SLC, MSN) and 4.3%(MLC, DAP), respectively. PES_IPC increases the write latency by 3.6% and 2.0% on average and at-most 6.9% (SLC, MSN) and 4.3%(MLC, DAP).

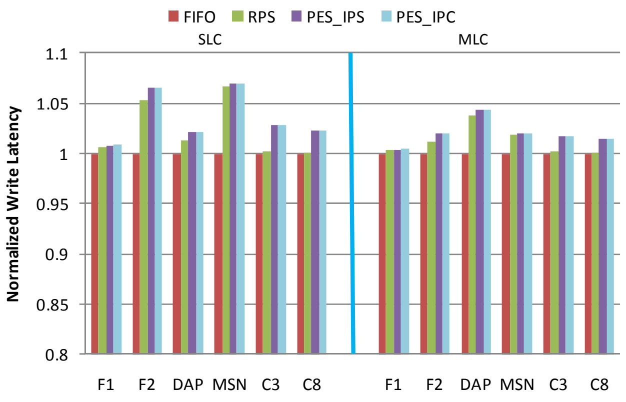


Figure 4: Write Latency Performance Comparison: FIFO, RPS, PES_IPC, PES_IPS. Normalized to FIFO.

Two major factors determine the write latency overhead: increased latency of each suspended P/E operation; the percentage of P/E that are suspended. We compare the original P/E latency reported by the device with latency after suspension in Fig. 5. The average overhead of suspended P/E is about 10.2% (SLC) and 7.8% (MLC). The percentage of suspended P/E is presented in Fig. 6. There is 4.9% (SLC) and 7.4% (MLC) of P/E's that are suspended on average. These two sets of results explain the low write overhead our design achieves.

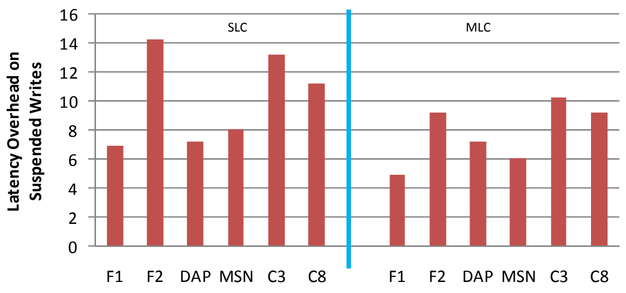


Figure 5: Compare the original write latency with the effective write latency resulted from P/E Suspension. Y axis represents the percentage of increased latency caused by P/E suspension.

5.3 Sensitivity Study on Write Queue Size

Finally, we study the sensitivity of write overhead to the write queue size. In order to obtain an amplified write overhead, we select F2, which has the highest percentage of read requests, and compress the simulation time

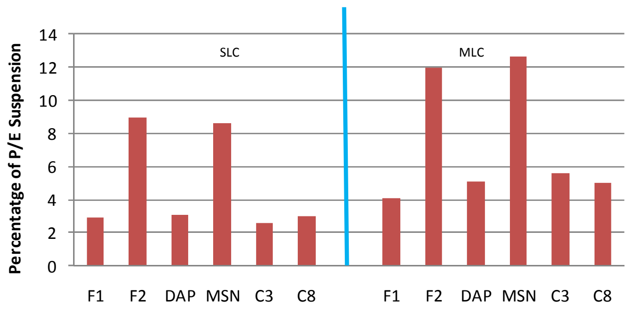


Figure 6: Percentage of suspended writes.

of F2 by 7 times to intensify the workload. In Figure 7 we present the write latency results of RPS and PES_IPC (normalized to that of FIFO) by varying the maximum write queue size from 16 to 512. Clearly, the write overhead of both RPS and PES_IPC is sensitive to the maximum write queue size, which suggests that the flash controller should limit the write queue size to control the write overhead. Noting that, relative to RPS, the PES_IPC has a near-constant increase on the write latency, which implies that the major contributor of overhead is RPS when the queue size varies.

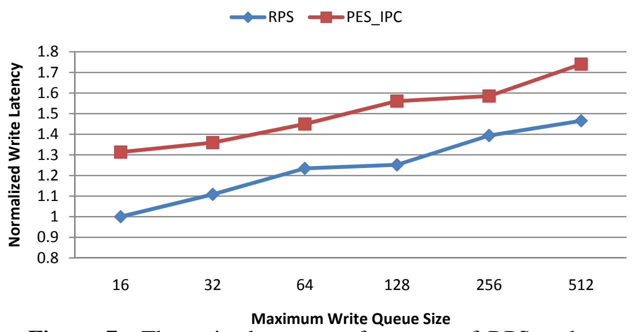


Figure 7: The write latency performance of RPS and PES_IPC while the maximum write queue size varies. Normalized to FIFO.

6 Conclusion and Future Work

One performance problem of NAND flash is that its program and erase latency is much higher than the read latency. This problem causes the chip contention between reads and P/Es due to the fact that with current NAND flash interface, the on-going P/E cannot be suspended and resumed. To alleviate the impact of the chip contention on the read performance, in this paper we propose a light-overhead *P/E suspension* scheme by exploiting the internal mechanism of P/E algorithm in NAND flash. The design is simulated/evaluated with precise timing and realistic SSD modeling of multi-chip/channel. Experimental results show that the proposed P/E suspension significantly reduces the read latency with trivial overhead on write performance.

Our future work will apply the idea of P/E suspension to further improve the performance of foreground processes via suspending the background operations (e.g., the garbage collection operations) in SSDs.

Acknowledgments

We thank Tong Zhang for the preliminary discussion on some physical parameters of NAND flash. We also thank our shepherd Dushyanth Narayanan and anonymous reviewers for their feedback. This research is sponsored in part by the U.S. National Science Foundation (NSF) under grants CCF-1102605 and CCF-1102624.

References

[1] AGRAWAL, N., PRABHAKARAN, V., AND ET AL. Design Tradeoffs for SSD Performance. In *USENIX ATC* (Boston, Massachusetts, USA, 2008).

[2] ARASE, K. Semiconductor NAND Type Flash Memory with Incremental Step Pulse Programming, Sept. 22 1998. U.S. Patent 5,812,457.

[3] BREWER, J., AND GILL, M. Nonvolatile Memory Technologies with Emphasis on Flash. *IEEE Whiley-Interscience, Berlin* (2007).

[4] DIMITRIJEVIC, Z., RANGASWAMI, R., AND CHANG, E. Design and Implementation of Semi-preemptible IO. In *FAST* (2003), USENIX, pp. 145–158.

[5] HEWLETT-PACKARD LABORATORIES. Cello99 Traces. `http://tesla.hpl.hp.com/opensource/`.

[6] JO, H., KANG, J.-U., AND ET AL. FAB: Flash-Aware Buffer Management Policy for Portable Media Players. *IEEE Transactions on Consumer Electronics 52*, 2 (2006), 485–493.

[7] KANG, S., AND ET AL. Performance Trade-Offs in Using NVRAM Write Buffer for Flash Memory-Based Storage Devices. *IEEE Transactions on Computers 58*, 6 (2009), 744–758.

[8] KAVALANEKAR, S., AND ET AL. Characterization of Storage Workload Traces from Production Windows Servers. In *IISWC* (2008).

[9] KIM, H., AND AHN, S. BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage Abstract. In *FAST* (2008).

[10] KIM, Y., ORAL, S., SHIPMAN, G., LEE, J., DILLOW, D., AND WANG, F. Harmonia: A Globally Coordinated Garbage Collector for Arrays of Solid-State Drives. In *MSST* (2011), IEEE, pp. 1–12.

[11] NARAYANAN, D., THERESKA, E., DONNELLY, A., ELNIKETY, S., AND ROWSTRON, A. Migrating server storage to SSDs: Analysis of tradeoffs. In *EuroSys* (2009).

[12] ONFI WORKING GROUP. The Open NAND Flash Interface, 2011. `http://onfi.org/`.

[13] QURESHI, M., AND ET AL. Improving Read Performance of Phase Change Memories via Write Cancellation and Write Pausing. In *HPCA* (2010), IEEE, pp. 1–11.

[14] STORAGE PERFORMANCE COUNCIL. SPC Trace File Format Specification. `http://traces.cs.umass.edu/index.php/Storage/Storage`.

[15] WU, G., HE, X., XIE, N., AND ZHANG, T. DiffECC: Improving SSD Read Performance Using Differentiated Error Correction Coding Schemes. *MASCOTS* (2010), 57–66.

Optimizing NAND Flash-Based SSDs via Retention Relaxation

Ren-Shuo Liu*, Chia-Lin Yang*, and Wei Wu†
*National Taiwan University and †Intel Corporation
{renshuo@ntu.edu.tw, yangc@csie.ntu.edu.tw, wei.a.wu@intel.com}

Abstract

As NAND Flash technology continues to scale down and more bits are stored in a cell, the raw reliability of NAND Flash memories degrades inevitably. To meet the retention capability required for a reliable storage system, we see a trend of longer write latency and more complex ECCs employed in an SSD storage system. These greatly impact the performance of future SSDs. In this paper, we present the first work to improve SSD performance via retention relaxation. NAND Flash is typically required to retain data for 1 to 10 years according to industrial standards. However, we observe that many data are overwritten in hours or days in several popular workloads in datacenters. The gap between the specification guarantee and actual programs' needs can be exploited to improve write speed or ECCs' cost and performance. To exploit this opportunity, we propose a system design that allows data to be written in various latencies or protected by different ECC codes without hampering reliability. Simulation results show that via write speed optimization, we can achieve 1.8–5.7× write response time speedup. We also show that for future SSDs, retention relaxation can bring both performance and cost benefits to the ECC architecture.

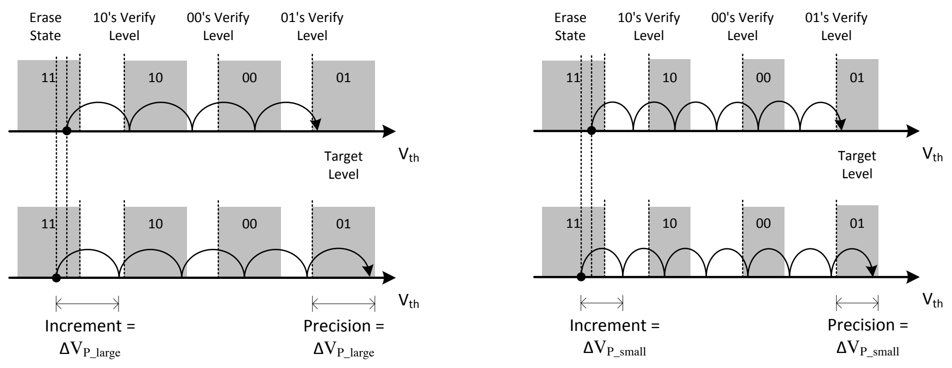
1 Introduction

For the past few years, NAND Flash memories have been widely used in portable devices such as media players and mobile phones. Due to their high density, low power and high I/O performance, in recent years, NAND Flash memories begun to make the transition from portable devices to laptops, PCs and datacenters [6, 35]. As the semiconductor industry continues scaling memory technology and lowering per-bit cost, NAND Flash is expected to replace the role of hard disk drives and fundamentally change the storage hierarchy in future computer systems [14, 16].

A reliable storage system needs to provide a retention guarantee. Therefore, Flash memories have to meet the retention specification in industrial standards. For example, according to the JEDEC standard JESD47G.01 [19], NAND Flash blocks cycled to 10% of the maximum specified endurance must retain data for 10 years, and blocks cycled to 100% of the maximum specified endurance have to retain data for 1 year. As NAND Flash technology continues to scale down and more bits are stored in a cell, the raw reliability of NAND Flash decreases substantially. To meet the retention specification for a reliable storage system, we see a trend of longer write latency and more complex ECCs required in SSDs. For example, comparing recent 2-bit MLC NAND Flash memories with previous SLC ones, page write latency increased from 200 μs [34] to 1800 μs [39], and the required strength of ECCs went from single-error-correcting Hamming codes [34] to 24-error-correcting Bose-Chaudhuri-Hocquenghem (BCH) codes [8, 18, 28]. In the near future, more complex ECC codes such as low-density parity-check (LDPC) [15] codes will be required to reliably operate NAND Flash memories [13, 28, 41].

To overcome the design challenge for future SSDs, in this paper, we present *retention relaxation*, the first work on optimizing SSDs via relaxing NAND Flash's retention capability. We observe that in typical datacenter workloads, e.g., proxy and MapReduce, many data written into storage are updated quite soon, thereby, requiring only days or even hours of data retention, which is much shorter than the retention time typically specified for NAND Flash. In this paper, we exploit the gap between the specification guarantee and actual programs' needs for SSD optimization. We make the following contributions:

- We propose a NAND Flash model that captures the relationship between raw bit error rates and retention time based on empirical measurement data. This model allows us to explore the interplay between retention capability and other NAND Flash parameters such as the program step voltage for write operations.
- A set of datacenter workloads are characterized for their retention time requirements. Since I/O traces are usually gathered in days or weeks, to analyze retention time requirements in a time span beyond the trace period, we present a retention time projection method based on two characteristics obtained from the traces, the write amount and the write working set size. Characterization results show that for 15 of the 16 traces analyzed, 49–99% of writes require less than 1-week retention time.
- We explore the benefits of retention relaxation for


Erase State
10's Verify Level
00's Verify Level
01's Verify Level
11
10
00
01
Target Level
V_{th}
Increment = ΔV_{P_large}
Precision = ΔV_{P_large}
Increment = ΔV_{P_small}
Precision = ΔV_{P_small}

(a) ISPP with large ΔV_P

(b) ISPP with small ΔV_P

Figure 1: Incremental step pulse programming (ISPP) for programming NAND Flash

speeding up write operations. We increase the program step voltage so that NAND Flash memories are programmed faster but with shorter retention guarantees. Experimental results show that 1.8–5.7× SSD write response time speedup is achievable.

- We show how retention relaxation can benefit ECC designs for future SSDs which require concatenated BCH-LDPC codes. We propose an ECC architecture where data are encoded by variable ECC codes based on their retention requirements. In our ECC architecture, time-consuming LDPC is removed from the critical performance path. Therefore, retention relaxation can bring both performance and cost benefits to the ECC architecture.

The rest of the paper is organized as follows. Section 2 provides background about NAND Flash. Section 3 presents our NAND Flash model and the benefits of retention relaxation. Section 4 analyzes data retention requirements in real-world workloads. Section 5 describes the proposed system designs. Section 6 presents evaluation results regarding the designs in the previous section. Section 7 describes related work, and Section 8 concludes the paper.

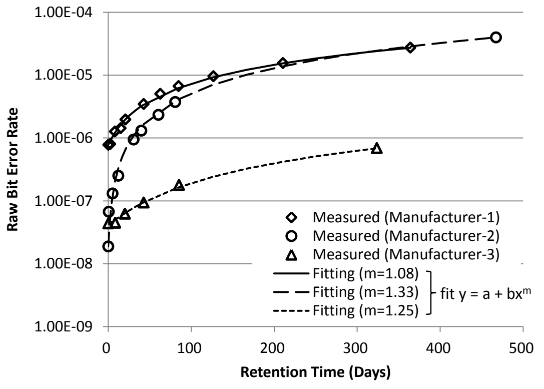
2 Background

NAND Flash memories comprise an array of floating gate transistors. The threshold voltage (V_{th}) of the transistors can be programmed to different levels by injecting different amounts of charge on the floating gates. Different V_{th} levels represent different data. For example, to store N bits data in a cell, its V_{th} is programmed to one of its 2^N different V_{th} levels.

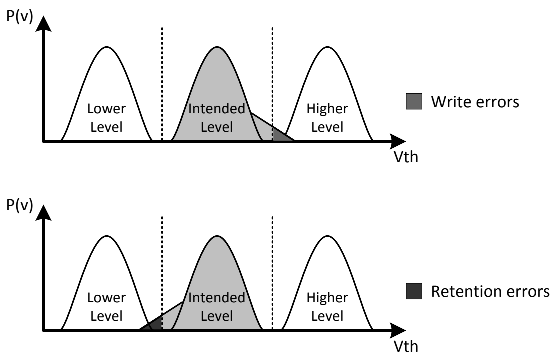
To program V_{th} to the desired level, the incremental step pulse programming (ISPP) scheme is commonly used [26, 37]. As shown in Figure 1, ISPP increases the V_{th} of NAND Flash cells step-by-step by a certain voltage increment (i.e., ΔV_P) and stops once V_{th} is greater than the desired threshold voltage. Because NAND Flash cells have different starting V_{th}, the resulting V_{th} spreads across a range, which determines the precision of cells' V_{th} distributions. The smaller ΔV_P is, the more precise the resulting V_{th} is. On the other hand, smaller ΔV_P means more steps are required to reach the target V_{th}, thereby, resulting in longer write latency [26].

NAND Flash memories are prone to errors. That is, the V_{th} level of a cell may be different from the intended one. The fraction of bits which contain incorrect data is referred to as the raw bit error rate (RBER). Figure 2(a) shows measured RBER of 63–72nm 2-bit MLC NAND Flash memories under room temperature following 10K program/erase (P/E) cycles [27]. The RBER at retention time = 0 is attributed to write errors. Write errors have been shown mostly caused by cells with higher V_{th} than intended because the causes of write errors, such as program disturb and random telegraph noise, tend to over-program V_{th}. The increment of RBER after writing data (retention time > 0) is attributed to retention errors. Retention errors are caused by charge losses which decrease V_{th}. Therefore, retention errors are dominated by cells with lower V_{th} than intended. Figure 2(b) illustrates these two error sources: write errors mainly correspond to the tail at the high-V_{th} side; retention errors correspond to the tail at the low-V_{th} side. In Section 3.1, we model NAND Flash considering these error characteristics.

A common approach to handle NAND Flash errors is to adopt ECCs (error correction codes). ECCs supplement user data with redundant parity bits to form codewords. With ECC protection, a codeword with a certain amount of bits corrupted can be reconstructed. Therefore, ECCs can greatly reduce the bit error rate. We refer to the bit error rate after applying ECCs as the uncorrectable bit error rate (UBER). The following equation gives the relationship between UBER and RBER [27]:



(a) Measured $RBER(t)$ and fitting to power-law trends for 63–72 nm 2-bit MLC NAND Flash. Data are aligned to $t = 0$.



(b) Write errors and retention errors

Figure 2: Bit error rate in NAND Flash memories

$$UBER = \frac{\sum_{n=t+1}^{N_{CW}} \binom{N_{CW}}{n} \cdot RBER^n \cdot (1-RBER)^{(N_{CW}-n)}}{N_{User}} \tag{1}$$

Here, N_{CW} is the number of bits per codeword, N_{User} is the number of user data bits per codeword, and t is the maximum number of error bits the ECC code can correct per codeword.

UBER is an important reliability metric for storage systems and is typically required to be under 10^{-13} to 10^{-16} [27]. As mentioned earlier, NAND Flash's RBER increases with time due to retention errors. Therefore, to satisfy both the retention and reliability specifications in storage systems, ECCs must be strong enough to tolerate not only write errors presenting in the beginning but also retention errors accumulating over time.

3 Retention Relaxation for NAND Flash

The key observation we make in this paper is that since retention errors increase over time, if we could relax the retention capability of NAND Flash memories, fewer retention errors need to be tolerated. These error margins can then be utilized to improve other performance metrics. In this section, we first present a V_{th} distribution modeling methodology which captures the RBER of NAND Flash. Based on the model, we elaborate on the strategies to exploit the benefits of retention relaxation in detail.

3.1 Modeling Methodology

We first present the base V_{th} distribution model for NAND Flash. Then we present how we extend the model to capture the characteristics of different error causes. Last, we determine the parameters of the model by fitting the model to the error-rate behavior of NAND Flash.

3.1.1 Base V_{th} Distribution Model

The V_{th} distribution is critical to NAND Flash. It describes the probability density function (PDF) of V_{th} for each data state. Given a V_{th} distribution, one can evaluate the corresponding RBER by calculating the probability that a cell contains incorrect V_{th}, i.e., V_{th} higher or lower than the intended level.

V_{th} distributions have been modeled using bell-shape functions in previous studies [23, 42]. For MLC NAND Flash memories with q states per cell, q bell-shape functions, $P_k(v)$ where $0 \le k \le (q-1)$, are employed in the model as follows.

First, the V_{th} distribution of the erased state is modeled as a Gaussian function, $P_0(v)$:

$$P_0(v) = \alpha_0 \cdot e^{-\frac{(v-\mu_0)^2}{2\sigma_0^2}} \tag{2}$$

Here, σ_0 is the standard deviation of the distribution and μ_0 is the mean. Because data are assumed to be in one of the q states with equal probability, a normalization coefficient, α_0, is employed so that $\int_v P_0(v) = \frac{1}{q}$.

Furthermore, the V_{th} distribution of each non-erased state (i.e., $1 \le k \le (q-1)$) is modeled as a combination of a uniform distribution with width equal to ΔV_P in the middle and two identical Gaussian tails on both sides:

$$P_k(v) = \begin{cases} \alpha \cdot e^{-\frac{(v-\mu_k+0.5\Delta V_P)^2}{2\sigma^2}}, & v < \mu_k - \frac{\Delta V_P}{2} \\ \alpha \cdot e^{-\frac{(v-\mu_k-0.5\Delta V_P)^2}{2\sigma^2}}, & v > \mu_k + \frac{\Delta V_P}{2} \\ \alpha, & \text{otherwise} \end{cases} \tag{3}$$

Here, ΔV_P is the voltage increment in ISPP, μ_k is the mean of each state, σ is the standard deviation of the two Gaussian tails, and α is again the normalization coefficient to satisfy the condition that $\int_v P_k(v) = \frac{1}{q}$ for the $k-1$ states.

Given the V_{th} distribution, the RBER can be evaluated by calculating the probability that a cell contains incorrect V_{th}, i.e., V_{th} higher or lower than the intended read voltage levels, using the following equation:

$$RBER = \sum_{k=0}^{q-1} \left(\underbrace{\int_{-\infty}^{V_{R,k}} P_k(v)dv}_{V_{th}\text{lower than intended}} + \underbrace{\int_{V_{R,(k+1)}}^{\infty} P_k(v)dv}_{V_{th}\text{higher than intended}} \right) \quad (4)$$

Here, $V_{R,k}$ is the lower bound of the correct read voltage for the k^{th} state and $V_{R,k+1}$ is the upper bound as shown in Figure 3.

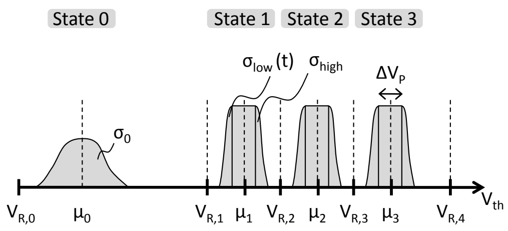


Figure 3: Illustration of model parameters

3.1.2 Model Extension

As mentioned in Section 2, the two tails of a V_{th} distribution are from different causes. The high-V_{th} tail of a distribution is mainly caused by V_{th} over-programming (i.e., write errors); the low-V_{th} tail is mainly due to charge losses over time (i.e., retention errors) [27]. Therefore, the two Gaussian tails may not be identical. To capture this difference, we extend the base model by setting different standard deviations to the two tails as shown in Figure 3.

The two standard deviations are set based on the observation in the previous study on Flash's retention process [7]. Under room temperature[1], a small portion of cells have a much larger charge-loss rate than others. As such charge losses accumulate over time, the distribution tends to form a wider tail at the low-V_{th} side. Therefore, we extend the base model by setting the standard deviation of the low-V_{th} tail to be a time-increasing function, $\sigma_{low}(t)$, but keeping σ_{high} time-independent. The extended model is as follows:

$$P_k(v,t) = \begin{cases} \alpha(t) \cdot e^{-\frac{(v-\mu_k+0.5\Delta V_P)^2}{2\sigma_{low}(t)^2}}, & v < \mu_k - \frac{\Delta V_P}{2} \\ \alpha(t) \cdot e^{-\frac{(v-\mu_k-0.5\Delta V_P)^2}{2\sigma_{high}^2}}, & v > \mu_k + \frac{\Delta V_P}{2} \\ \alpha(t), & \text{otherwise} \end{cases} \quad (5)$$

Here, the normalization term becomes a function of time, $\alpha(t)$, to keep $\int_v P_k(v,t) = \frac{1}{q}$.

[1]According to the previous study [32], in datacenters, HDDs' average temperatures range between 18–51°C and stay around 26–30°C most of the time. Since SSDs do not contain motors and actuators, we expect SSDs should be in lower temperature than HDDs. Therefore, we only consider room temperature in our current model.

We should note that keeping σ_{high} time-independent does not imply that cells with high V_{th} are time-independent and never leak charge. Since the integral of PDF for each data state remains $\frac{1}{q}$, the probability that a cell belongs to the high-V_{th} tail drops as the low-V_{th} tail widens over time. The same phenomenon happens to the middle part, too.

Given the V_{th} distribution in the extended model, $RBER(t)$ can be evaluated using the following formula:

$$RBER(t) = \sum_{k=0}^{q-1} \left(\underbrace{\int_{-\infty}^{V_{R,k}} P_k(v,t)dv}_{V_{th}\text{lower than intended}} + \underbrace{\int_{V_{R,(k+1)}}^{\infty} P_k(v,t)dv}_{V_{th}\text{higher than intended}} \right) \quad (6)$$

3.1.3 Model Parameter Fitting

In the proposed V_{th} distribution model, ΔV_P, $V_{R,k}$, μ_k, and σ_0 are set to the values shown in Figure 4 according to [9]. The two new parameters in the extended model, σ_{high} and $\sigma_{low}(t)$, are determined through parameter fitting such that the resulting $RBER(t)$ follows the error-rate behavior of NAND Flash. Below we describe the parameter fitting procedure.

We adopt the power-law model [20] to describe the error-rate behavior of NAND Flash:

$$RBER(t) = RBER_{write} + RBER_{retention} \times t^m \quad (7)$$

Here, t is time, m is a coefficient, $1 \leq m \leq 2$, $RBER_{write}$ corresponds to the error rate at $t = 0$ (i.e., write errors), and $RBER_{retention}$ is the incremental error rate per unit of time due to retention errors.

We determine m in the power-law model based on the curve-fitting values shown in Figure 2(a). In the figure, the power-law curves fit the empirical error-rate data very well with m equal to 1.08, 1.25, and 1.33. We consider 1.25 as the typical case of m and consider the other two values as the corner cases.

The other two coefficients in the power-law model, $RBER_{write}$ and $RBER_{retention}$, can be solved given RBER at $t = 0$ and RBER at the maximum retention time, t_{max}. According to the JEDEC standard JESD47G.01 [19], NAND Flash blocks cycled to the maximum specified endurance have to retain data for 1 year, so we set t_{max} to 1 year. Moreover, recent NAND Flash requires 24-bit error correction for 1080-byte data [4, 28]. Assuming that the target $UBER(t_{max})$ requirement is 10^{-16}, by Equation (1), we have:

$$RBER(t_{max}) = 4.5 \times 10^{-4} \quad (8)$$

As shown in Figure 2(a), $RBER(0)$ is typically orders of magnitude lower than $RBER(t_{max})$. Tanakamaru *et al.* [41] also show that write errors are between 150× to 450× fewer than retention errors. This is because retention errors accumulate over time and eventually dominate. Therefore, we set $RBER_{write}$ accordingly:

$$RBER_{write} = RBER(0) = \frac{RBER(t_{max})}{C_{write}} \quad (9)$$

Parameter	Value (V)
$V_{R,0}$	$-\infty$
$V_{R,1}$	0
$V_{R,2}$	1.2
$V_{R,3}$	2.4
$V_{R,4}$	6
ΔV_P	0.2
μ_0	-3
μ_1	0.6
μ_2	1.8
μ_3	3
σ_0	0.48
σ_{high}	0.1119
$\sigma_{low}(0)$	0.1039

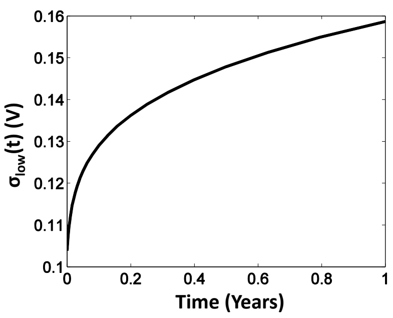


Figure 4: Modeled 2-bit MLC NAND Flash

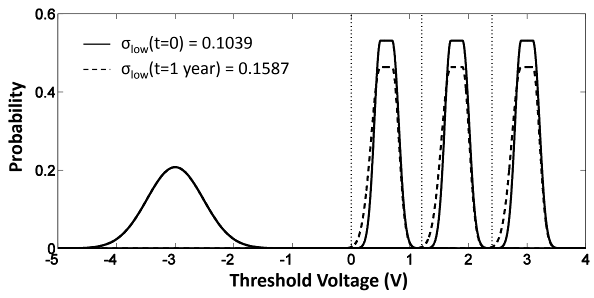


Figure 5: Modeling results

Here, C_{write} is the ratio of $RBER(t_{max})$ to $RBER(0)$. We set C_{write} to 150, 300, and 450, where 300 is considered as the typical case and the other two are considered as the corner cases.

We also have $RBER_{retention}$ as follows:

$$RBER_{retention} = \frac{(RBER(t_{max}) - RBER(0))}{{t_{max}}^m} \tag{10}$$

We note that among write errors (i.e., $RBER_{write}$), a major fraction of them correspond to cells with higher V_{th} than intended. This is because the root causes of write errors tend to make V_{th} over-programmed. Mielke *et al.* [27] show that this fraction is between 62% to about 100% for the NAND Flash devices in their experiments. Therefore, we give the following equations:

$$RBER_{write\_high} = RBER_{write} \times C_{write\_high} \tag{11}$$

$$RBER_{write\_low} = RBER_{write} \times (1 - C_{write\_high}) \tag{12}$$

Here, $RBER_{write\_high}$ and $RBER_{write\_low}$ correspond to cells with V_{th} higher and lower than intended levels, respectively. C_{write_high} stands for the ratio of total write errors to write errors which are higher than the intended levels. We set C_{write_high} to 62%, 81%, and 99%, where 81% is considered as the typical case and the other two are considered as the corner cases.

Now we have the error-rate behavior of NAND Flash. σ_{high} and $\sigma_{low}(0)$ are first determined so that the error rate for V_{th} being higher and lower than intended equals $RBER_{write\_high}$ and $RBER_{write\_low}$, respectively. Then, $\sigma_{low}(t)$ is determined by matching the $RBER(t)$ derived from the V_{th} model with NAND Flash's error-rate behavior described in Equations (7) to (10) at a fine time step.

Figure 5 shows the modeling results of the V_{th} distribution for the typical-case NAND Flash. In this figure, the solid line stands for the V_{th} distribution at $t = 0$; the dashed line stands for the V_{th} distribution at $t = 1$ year. We can see that the 1-year distribution is flatter than the distribution at $t = 0$. We can also see that as the low-V_{th} tail widens over a year, the probability of both the middle part and the high-V_{th} tail drops correspondingly.

3.2 Benefits of Retention Relaxation

In this section, we elaborate on the benefits of retention relaxation from two perspectives — improving write speed and improving ECCs' cost and performance. The analysis is based on NAND Flash memories cycled to the maximum specified endurance (i.e., 100% wear-out) with data retention capability set to 1 year [19]. Since NAND Flash's reliability typically degrades monotonically in terms of P/E cycles, considering such an extreme case is conservative for the following benefit evaluation. In other words, NAND Flash in its early lifespan has more head room for optimization.

3.2.1 Improving Write Speed

As presented earlier, NAND Flash memories use the ISPP scheme to incrementally program memory cells. The V_{th} step increment, ΔV_P, directly affects write speed and data retention. Write speed is proportional to ΔV_P because with larger ΔV_P, less steps are required during the ISPP procedure. On the other hand, data retention decreases as ΔV_P gets larger because large ΔV_P widens V_{th} distributions and reduces the margin for tolerating retention errors.

Algorithm 1 shows the procedure to quantitatively evaluate the write speedup if data retention time requirements are reduced. The analysis is based on the extended NAND Flash model presented in Section 3.1. For all the typical and corner cases we consider, we first enlarge ΔV_P by various ratios between $1\times$ to $3\times$, thereby, speeding up NAND Flash writes proportionately. For each ratio, we test $RBER(t)$ at different retention time from 0 to 1 year to find the maximum t such that $RBER(t)$ is within the base ECC strength.

Figure 6 shows the write speedup vs. data retention. Both the typical case (black line) and the corner cases (gray dashed lines) we consider in Section 3.1 are shown. For the typical case, if data retention is relaxed to 10 weeks, $1.86\times$ speedup for NAND Flash page write is achievable; if data retention is relaxed to 2 weeks, the speedup is $2.33\times$. Furthermore, the speedup for the corner cases are close to the typical case. This means the speedup numbers are not very sensitive to the values of the parameters we obtain using parameter fitting.

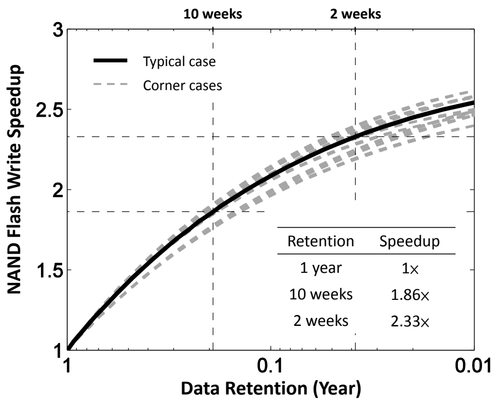


Figure 6: NAND page write speedup vs. data retention

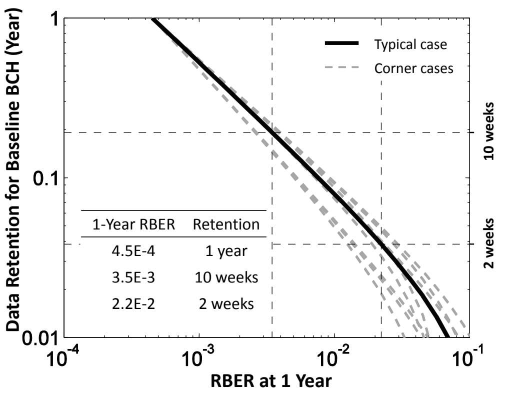


Figure 7: Data retention capability of 24 error correction per 1080 bytes BCH codes

Algorithm 1 Write speedup vs. data retention

```
 1: $C_{BCH} = 4.5 \times 10^{-4}$
 2: for all typical and corner cases do
 3:     for $VoltageRatio = 1$ to 3 step = 0.01 do
 4:         Enlarge $\Delta V_P$ by $VoltageRatio$ times
 5:         $WriteSpeedUp = VoltageRatio$
 6:         for $Time\ t = 0$ to 1 $year$ step = $\delta$ do
 7:             Find $RBER(t)$ according to $\sigma_{low}(t)$ and $\alpha(t)$
 8:         end for
 9:         $DataRetention = max\{t{:}RBER(t) \le C_{BCH}\}$
10:         plot $(DataRetention, WriteSpeedUp)$
11:     end for
12: end for
```

3.2.2 Improving ECCs' Cost and Performance

ECC design is emerging as a critical issue in SSDs. Nowadays, NAND Flash-based systems heavily rely on BCH codes to tolerate RBER. Unfortunately, BCH degrades memory storage efficiency significantly once the RBER of NAND Flash reaches 10^{-3} [22]. Recent NAND Flash has RBER around 4.5×10^{-4}. As the density of NAND Flash memories continues to increase, RBER will exceed the BCH limitation inevitably. Therefore, BCH codes will become inapplicable in the near future.

LDPC codes are promising ECCs for future NAND Flash memories [13, 28, 41]. The main advantage of LDPC is that they can provide correction performance very close to the theoretical limits. However, LDPC incurs much higher encoding complexity than BCH does [21, 25]. For example, an optimized LDPC encoder [44] consumes 3.9 M bits of memory and 11.4 k FPGA Logic Elements to offer 45 MB/s throughput. To sustain write throughput of high-performance SSDs, e.g., 1 GB/s ones [1], high-throughput LDPC encoders are required, otherwise the LDPC encoders may become the throughput bottleneck. This leads to high hardware cost because hardware parallelization is one basic approach to increase the throughput of LDPC encoders [24]. In this paper, we exploit retention relaxation to alleviate such cost and performance dilemma. That is, with retention relaxation, fewer retention errors need to be tolerated; therefore, BCH codes could be still strong enough to protect data even if NAND Flash's 1-year RBER soars.

Algorithm 2 analyzes the achievable data retention time of BCH codes with 24 bits per 1080 bytes error-correction capability under different NAND Flash $RBER(1\ year)$ values. Here we assume that $RBER(t)$ follows the power-law trend described in Section 3.1.3. We vary $RBER(1\ year)$ from 4.5×10^{-4} to 1×10^{-1}, and derive the corresponding write error rate ($RBER_{write}$) and retention error increment per unit of time ($RBER_{retention}$). The achievable data retention time of the BCH codes is the time when RBER exceeds the capability of the BCH codes (i.e., 4.5×10^{-4}).

Algorithm 2 Data retention vs. maximum RBER for BCH (24-bit per 1080 bytes)

```
 1: $t_{max} = 1\ year$
 2: $C_{BCH} = 4.5 \times 10^{-4}$
 3: for all typical and corner cases do
 4:     for $RBER(t_{max}) = 4.5 \times 10^{-4}$ to $1 \times 10^{-1}$ step = $\delta$ do
 5:         $RBER_{write} = \frac{RBER(t_{max})}{C_{write}}$
 6:         $RBER_{retention} = \frac{(RBER(t_{max}) - RBER_{write})}{{t_{max}}^m}$
 7:         $RetentionTime = (\frac{C_{BCH} - RBER_{write}}{RBER_{retention}})^{\frac{1}{m}}$
 8:         plot $(RBER(t_{max}), RetentionTime)$
 9:     end for
10: end for
```

Figure 7 shows the achievable data retention time of the BCH code given different $RBER(1\ year)$ values. The black line stands for the typical case and the gray dashed lines stand for the corner cases. As can be seen, for the typical case, the baseline BCH code can retain data for

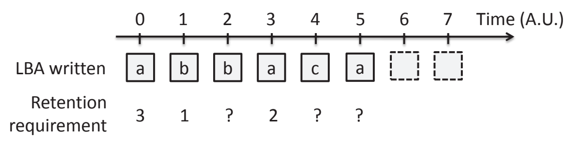


Figure 8: Data retention requirements in a write stream

10 weeks even if $RBER(1\ year)$ reach 3.5×10^{-3}. Even if $RBER(1\ year)$ reaches 2.2×10^{-2}, the baseline BCH code can still retain data for 2 weeks. We can also see similar trends for the corner cases.

4 Data Retention Requirements Analysis

In this section, we first analyze real-world traces from enterprise datacenters to show that many writes into storage require days or even shorter retention time. Since I/O traces are usually gathered in days or weeks, to estimate the percentage of writes with retention requirements beyond the trace period, we present a retention-time projection method based on two characteristics obtained from the traces, the write amount and the write working set size.

4.1 Real Workload Analysis

In this subsection, we analyze real disk traces to understand the data retention requirements of real-world applications. The data retention requirement of a sector written into a disk is defined as: *the interval from the time the sector is written to the time the sector is overwritten.* Let's take Figure 8 for example. The disk is written by the address stream **a, b, b, a, c, a, ...** and so on. The first write is to address **a** at time 0, and the same address is overwritten at time 3; therefore, the data retention requirement for the first write is $(3-0) = 3$. Usually disk traces only cover a limited period of time, for those writes whose next write does not appear before the observation ends, the retention requirements cannot be determined. For example, for the write to address **b** at time 2, the overwritten time is unknown. We denote its retention requirement with '?' as a conservative estimation. It is important to note that we are focusing on data retention requirements for data blocks *in write streams* rather than that *in the entire disk*.

Table 1 shows the three sets of traces we analyze. The first is from an enterprise datacenter in Microsoft Research Cambridge (MSRC) [29]. This set covers 36 volumes from various servers and we select 12 of them which have the largest write amounts. These traces span 1 week and 7 hours. We skip the first 7 hours which do not form a complete day and use the remaining 1-week part. The second set of traces is MapReduce which has been shown to benefit from the increased bandwidth and reduced latency of NAND Flash-based SSDs [11]. We use Hadoop [2] to run the MapReduce benchmark on a cluster of two Core-i7 machines each of which has

Table 1: Workload summary

Category	Name	Description	Span
MSRC	prn_0 proj_0, proj_2 prxy_0, prxy_1 src1_0, src1_2 src2_2 usr_1, usr_2	Print server Project directories Web proxy Source control Source control User home directories	1 week
MapReduce	hd1 hd2	WordCount benchmark	1 day
TPC-C	tpcc1 tpcc2	OLTP benchmark	1 day

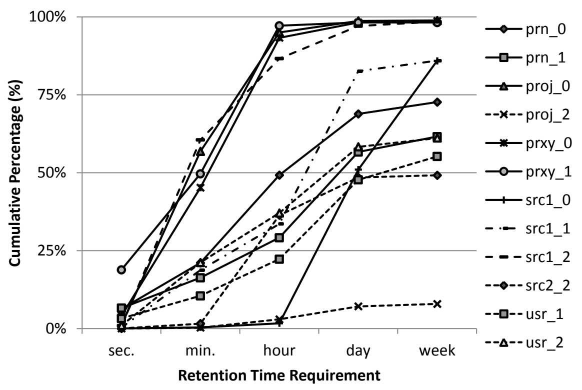
8 GB RAM and a SATA hard disk and runs 64-bit Linux 2.6.35 with the ext4 filesystem. We test two MapReduce usage models. In the first model, we repeatedly replace 140 GB text data in the Hadoop cluster and invoke word counting jobs. In the second model, we interleave performing word counting jobs on two sets of 140 GB text data which have been pre-loaded in the cluster. The third workload is the TPC-C benchmark. We use Hammerora [3] to generate the TPC-C workload on a MySql server which has a Core-i7 CPU, 12 GB RAM, and a SATA SSD and runs 64-bit Linux 2.6.32 with the ext4 filesystem. We configure the benchmarks as having 40 and 80 warehouses. Each warehouse has 10 users with keying and thinking time. Both MapReduce and TPC-C workloads span 1 day.

For each trace, we analyze the data retention requirement of every sector written into the disk. Figure 9 shows the cumulative percentage of data retention requirements less than or equal to the following values — a second, a minute, an hour, a day, and a week. As can be seen, the data retention requirements of the workloads are usually low. For example, more than 95% of sectors written into the disk for proj_0, prxy_1, tpcc1, and tpcc2 need less than 1-hour data retention. Furthermore, for all the traces except proj_2, 49–99.2% of sectors written into the disk need less than 1-week data retention. For tpcc2, up to 44% of writes require less than 1-second retention. This is because MySql's storage engine, InnoDB, writes data to a fixed-size log, called the doublewrite buffer, before writing to the data file to guard against partial page writes; therefore, all writes to the doublewrite buffer are overwritten very quickly.

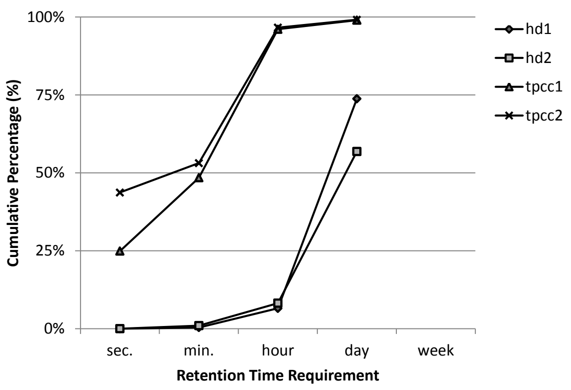
4.2 Retention Requirement Projection

The main challenge of retention time characterization for real-world workloads is that I/O traces are usually gathered in a short period of time, e.g., days or weeks. To estimate the percentage of writes with retention requirements beyond the trace period, we derive a projection method based on two characteristics obtained from the traces, the write amount and the write working set size.

We denote the percentage of writes with retention time



(a) MSRC workloads



(b) MapReduce and TPC-C workloads

Figure 9: Data retention requirement distribution

requirements less than X within a time period of Y as $S_{X,Y}\%$. We first formulate $S_{T_1,T_1}\%$ in the write amount and the write working set size, where T_1 is the time span of the trace. Let N be the amount of data sectors written into the disk during T_1 and W be the write working set size (i.e., the number of distinct sector addresses being written) during T_1. We have the following formula (the proof is similar to the pigeonhole principle):

$$S_{T_1,T_1}\% = \frac{N-W}{N} = 1 - \frac{W}{N} \quad (13)$$

With this formula, the first projection we make is the percentage of writes that have retention time requirements less than T_1 in an observation period of T_2, where $T_2 = k \times T_1$, $k \in \mathbb{N}$. The projection is based on the assumption that for each T_1 period, the write amount and the write working set size remain N and W, respectively. We derive the lower bound on $S_{T_1,T_2}\%$ as follows:

$$S_{T_1,T_2}\% = \frac{k(N-W) + \sum_{i=1}^{k-1} u_i}{kN} \geq S_{T_1,T_1}\% \quad (14)$$

where u_i is the number of sectors whose lifetime is across two periods and their retention time requirements are less than T_1. Equation (14) implies that we do not overestimate the percentage of writes that have retention time requirements less than T_1 by characterizing a trace gathered in a T_1 period.

With the first projection, we can then derive the lower bound on $S_{T_2,T_2}\%$. Clearly, $S_{T_2,T_2}\% \geq S_{T_1,T_2}\%$. Combined with Equation (14), we have:

$$S_{T_2,T_2}\% \geq S_{T_1,T_2}\% \geq S_{T_1,T_1}\% \quad (15)$$

The lower bound on $S_{T_2,T_2}\%$ also depends on disk capacity, A. During T_2, the write amount is equal to $k \times N$, and the write working set size must be less than or equal to the disk capacity, i.e, $k \times W \leq A$. By with Equation (13), we have:

$$S_{T_2,T_2}\% \geq \frac{kN-A}{kN} = 1 - \frac{A}{kN} \quad (16)$$

Combining Equation (15) and (16), the lower bound on $S_{T_2,T_2}\%$ is given by:

$$S_{T_2,T_2}\% \geq max(1 - \frac{A}{kN}, S_{T_1,T_1}\%) \quad (17)$$

Table 2 shows the data retention requirements analysis using the above equations. First, we can see that the $S_{T_1,T_1}\%$ obtained from Equation (13) matches Figure 9. Let's take hd2 for example. There are a total of 726 GB of writes in 1 day whose write working set size is 313 GB. According to Equation (13), 57% of the writes whose retention time requirements are less than 1 day. This is the case shown in Figure 9. Furthermore, if we can observe the hd_2 workload for 1 week, more than 86% of writes whose retention time requirements are expected to be less than 1 weeks. This again shows the gap between the specification guarantee and actual programs' needs in terms of data retention.

5 System Design

5.1 Retention-Aware FTL (Flash Translation Layer)

In this section, we present the SSD design which leverages retention relaxation for improving either write speed or ECCs' cost and performance. Specifically, in the proposed SSD, data written into NAND Flash memories could occur in variable write latencies or be encoded by different ECC codes, which provide different levels of retention guarantees. We refer to the data written by these different methods as in different "modes". In our design, data in a physical NAND Flash block are in the same mode. To correctly retrieve data from NAND Flash, we need to record the mode of each physical block. Furthermore, to avoid data losses due to a

Table 2: Data retention requirements analysis

Volume Name	Disk Capacity (A)	Write Amount (N)	Write Working Set (W)	$S_{1d,1d}$	$S_{1w,1w}$	$S_{5w,5w}$
	GB	GB	GB	%	%	%
prn_0	66.3	44.2	12.1		72.6	≧72.6
prn_1	385.2	28.8	11.1		61.6	≧61.6
proj_0	16.2	143.8	1.6		98.9	≧98.9
proj_2	816.2	168.4	155.1		7.9	≧7.9
prxy_0	20.7	52.7	0.7		98.7	≧98.7
prxy_1	67.8	695.3	12.5		98.2	≧98.2
src1_0	273.5	808.6	114.1		85.9	≧93.2
src1_1	273.5	29.6	4.2		85.9	≧85.9
src1_2	8.0	43.4	0.7		98.5	≧98.5
src2_2	169.6	39.3	20.0		49.2	≧49.2
usr_1	820.3	54.8	24.5		55.2	≧55.2
usr_2	530.4	25.6	10.0		61.1	≧61.1
hd1	737.6	1564.9	410.1	73.8	≧93.3	≧98.7
hd2	737.6	726.3	313.3	56.9	≧85.5	≧97.1
tpcc1	149	310.3	3.1	99.0	≧99.0	≧99.0
tpcc2	149	692.8	6.0	99.1	≧99.1	≧99.4

[1] The disk capacity of the MSRC traces are estimated using their maximum R/W address. The estimation results conform to the previous study [30].

[2] 1d, 1w, and 5w stand for a day, a week, and 5 weeks, respectively.

[3] GB stands for 2^{30} bytes.

shortage of data retention capability, we have to monitor the remaining retention capability of each NAND Flash block. We implement the proposed retention-aware design in the Flash Translation Layer (FTL) in SSDs rather than in OSes. FTL-based implementation requires minimum OS/application modification, which we think is important for easy deployment and wide adoption of the proposed scheme.

Figure 10 shows the block diagram of the proposed FTL. The proposed FTL is based on the page-level FTL [5] with two additional components, Mode Selector (MS) and Retention Tracker (RT). For writes, MS sends different write commands to NAND Flash chips or invokes different ECC encoders. As discussed in Section 3.2.1, write speed could be improved by adopting larger ΔV_P. In current Flash chips, only one write command is supported. To support the proposed mechanism, NAND Flash chips need to provide multiple write commands with different ΔV_P values. MS keeps the mode of each NAND Flash block in memories so that during reads, it can invoke the right ECC decoder to retrieve data. RT is responsible for ensuring that every NAND Flash block in the SSD does not run out of its retention capability. RT uses one counter per NAND Flash block to keep track of its remaining retention time. When the first page of a block is written, the retention capability of this write is stored in the counter. These retention counters are periodically updated. If a block is found to approach its data retention limit, RT schedules background operations to move valid data in this block to another new block and then invalidates the old one.

One main parameter in the proposed SSD design is how many write modes we should employ in the SSD. The optimal setting depends on retention time varia-

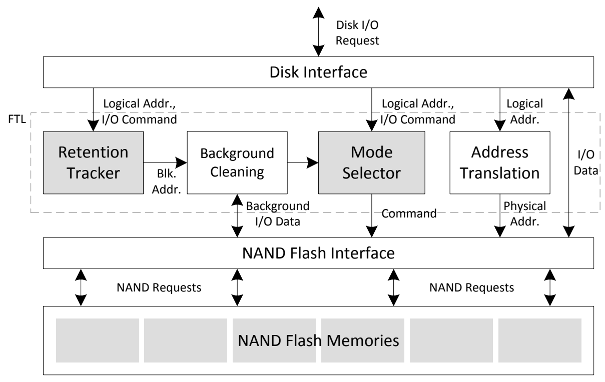


Figure 10: Proposed retention-aware FTL

tion in workloads and the cost for supporting multiple write modes. In this work, we present a coarse-grained management method. There are two kinds of NAND Flash writes in SSD systems: host writes and background writes. Host writes correspond to write requests sent from the host to the SSDs; background writes comprise cleaning, wear-leveling, and data movement internal to the SSDs. Performance is usually important to the host writes. Moreover, host writes usually require short data retention as shown in Section 4. In contrast, background writes are less sensitive to performance and usually involve data which have been stored in the storage for a long time; therefore, their data are expected to remain for a long time in the future (commonly referred to as cold data). Based on this observation, we propose to employ two levels of retention guarantees for the two kinds of writes. For host writes, retention-relaxed writes are used to exploit their high probability of short retention requirements and gain performance benefits; for background writes, normal writes are employed to preserve the retention guarantee.

In the proposed two-level framework, to optimize write performance, host writes occur in fast write speed with reduced retention capability. If data are not overwritten within their retention guarantee, background writes with normal write speed are issued. To optimize ECCs' cost and performance, a new ECC architecture is proposed. As mentioned earlier, NAND Flash RBER will soon exceed BCH's limitation (i.e., RBER $\geq 10^{-3}$); therefore, advanced ECC designs will be required for future SSDs. Figure 11 shows such an advanced ECC design for future SSDs which employs multi-layer ECCs with code concatenations: the inner code is BCH, and the outer code is LDPC. Concatenating BCH and LDPC exploits the advantages of both [43]: LDPC greatly improves the maximum correcting capability, while BCH complements LDPC for eliminating LDPC's error floor. The main issue with this design is since every write needs to be encoded in LDPC, a high-throughput LDPC encoder is required to prevent the LDPC encoder from be-

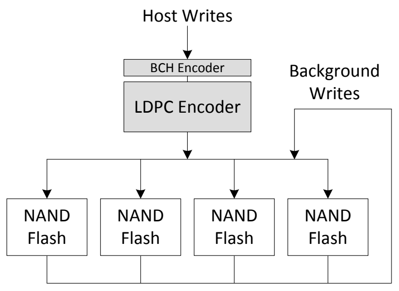


Figure 11: Baseline concatenated BCH-LDPC codes in an SSD

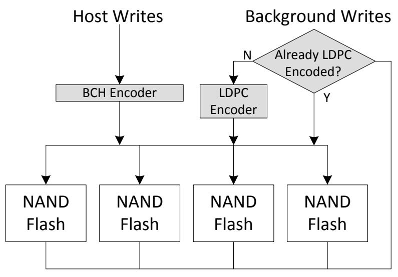


Figure 12: Proposed ECC architecture leveraging retention relaxation in an SSD

ing the bottleneck. In the proposed ECC architecture shown in Figure 12, host writes are protected by BCH only since they tend to have short retention requirements. If data are not overwritten within the retention guarantee provided by BCH, background writes are issued. All background writes are protected by LDPC. In this way, the LDPC encoder is kept out of the critical performance path. Its benefits are two-fold. First, write performance is improved since host writes do not go through time-consuming LDPC encoding. Second, since BCH filters out short-lifetime data and LDPC encoding can be amortized in the background, the throughput requirements of LDPC are less than the baseline design. Therefore, the LDPC hardware cost can be reduced.

We present two specific implementation of retention relaxation. The first one relaxes the retention capability of host writes to 10 weeks and periodically checks the remaining retention capability of each NAND Flash block at the end of every 5 weeks. Therefore, FTL always has another 5 weeks at least to reprogram those data which have not been overwritten in the past period and can amortize the re-programming task in the background over the 5 weeks without causing burst writes. We set the period of invoking the reprogramming tasks to 100 ms. The second one is similar to the first one except that the retention capability and checking period are 2 weeks and 1 week, respectively. These two designs are referred to as **RR-10week** and **RR-2week** in this paper.

5.2 Overhead Analysis

Memory Overhead

The proposed mechanism requires extra memory resources to store write modes and retention time information for each block. Since we only have two write modes, i.e., the normal mode and the retention-relaxed one, each block requires only a 1-bit flag to record its write mode. As for the size of the counter for keeping track of the remaining retention time, both RR-2week and RR-10week require only a 1-bit counter per block because all retention-relaxed blocks written in the n^{th} period are reprogrammed during the $(n+1)^{th}$ period. For

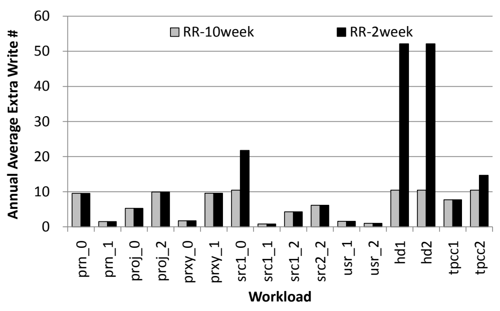


Figure 15: Wear-out overhead of retention relaxation

an SSD having 128 GB NAND Flash with 2 MB block size, the memory overhead is 16 KB .

Reprogramming Overhead

In the proposed schemes, data that are not overwritten in the guaranteed retention time need to be reprogrammed. These extra writes affect both the performance and the life time of SSDs. To analyze its performance impact, we estimate *reprogramming amounts per unit of time* based on the projection method described in Section 4.2. Here, we let T_2 be the checking period in the proposed schemes. For example, for RR-10week, T_2 equals 5 weeks. Therefore, at the end of each period, the total write amount is kN, the percentage of writes which require reprogramming is at most $(1 - S_{T_2,T_2}\%)$, and the reprogramming tasks can be amortized over the upcoming period of T_2. The reprogramming amounts per unit of time are as follows:

$$\frac{(1 - S_{T_2,T_2}\%) \times k \times N}{T_2} \tag{18}$$

The results show that the amount of reprogramming tasks range between 1.13 kB/s to 1.25 MB/s for RR-2week, and between 1.13 kB/s to 0.26 MB/s for RR-10week. Since each NAND Flash plane can provide 6.2 MB/s write throughput (i.e., writing a 8 kB page in 1.3 ms), we anticipate that reprogramming does not lead to high performance overhead. In Section 6, we evaluate its actual performance impact.

To quantify the wear-out effect caused by reprogramming, we show *extra writes per cell per year* assuming

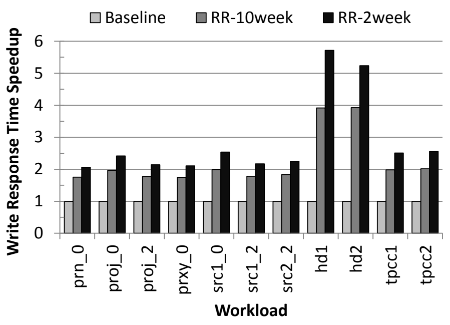


Figure 13: SSD write response time speedup

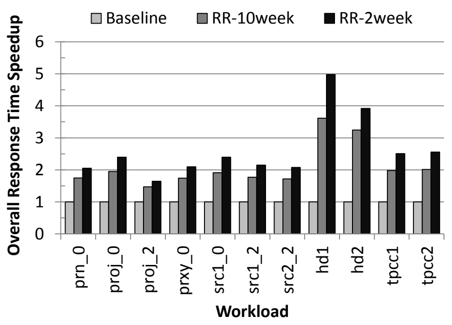


Figure 14: SSD overall response time speedup

perfect wear-leveling. We first give the upper bound on this metric. Let's take RR-2week for example. In the extreme case, RR-2week reprograms the entire disk every week, which leads to 52.1 extra writes per cell per year. Similarly, RR-10week causes 10.4 extra writes per cell per year at most. These extra writes are not significant compared to NAND Flash's endurance which is usually a few thousands P/E cycles. Therefore, even in the worst case, the proposed mechanism does not cause significant wear-out effect. For real workloads, the wear-out overhead is usually smaller than the worst case as shown in Figure 15. The wear-out overhead for each workload is evaluated based on the disk capacity and the reprogramming amounts per unit of time presented above.

6 System Evaluation

We conduct simulation-based experiments using SSDsim [5] and Disksim-4.0 [10] to evaluate the RR-10week and RR-2week designs. SSDs are configured to have 16 channels. Detailed configurations and parameters are listed in Table 3. Eleven of the 16 traces listed in Table 2 are used and simulated for the whole trace. We omit prxy_1 because the simulated SSD can not sustain its load, and prn_1, src1_1, usr_1, usr_2 are also omitted because they contain write amounts less than 15% of the total raw NAND Flash capacity. SSD write speedup and ECCs' cost and performance improvement are evaluated separately. The reprogramming overhead described in Section 5.2 are considered in the experiments.

Figure 13 shows the speedup of write response time for different workloads if we leverage retention relaxation to improve write speed. We can see that RR-10week and RR-2week typically achieve 1.8–2.6× write response time speedup. hd1 and hd2 show up to 3.9–5.7× speedup. These two workloads have high queuing delay due to high I/O throughput. With retention relaxation, the queuing time is greatly reduced, between 3.7× to 6.1×. Moreover, for all workloads, RR-2week gives about 20% extra performance gain over RR-10week. Figure 14 shows the speedup in terms of overall response time. The overall response time is mainly determined by write requests due to the significant amount of write requests in the tested workloads and the long write latency. Therefore, we can see that the speedup trend is similar to that of write response time.

Table 3: NAND Flash and SSD configurations

Parameter	Value
Over-provisioning	15%
Cleaning threshold	5%
Page size	8 KB
Pages per block	256
Blocks per plane	2000
Planes per die	2
Dies per channel	1~8
Number of channel	16
Mapping policy	Full stripe

Parameter	Value
Page read latency	75 µs
Page write latency	1.3 ms
Block erase latency	3.8 ms
NAND bus bandwidth	200 MB/s

Trace Name	Dies per Disk	Exported Capacity (GB)
prn_0, proj_0, prxy_0, src1_2	16	106
src2_2	32	212
src1_0	64	423
proj_2, hd1, hd2, tpcc1, tpcc2	128	847

To show how retention relaxation benefits ECC design in future SSDs, we consider SSDs comprising NAND Flash whose 1-year RBER approaches 2.2×10^{-2}. We compare the proposed RR-2week design with the baseline design which employs concatenated BCH-LDPC codes. The LDPC encoder is modeled as a FIFO and its throughput is chosen among 5, 10, 20, 40, 80, 160, 320, and ∞ MB/s. Since the I/O queue of the simulated SSDs could saturate if LDPC's throughput is insufficient, we first report the minimum required throughput configurations without causing saturation in Figure 16. As can be seen, for the baseline ECC architecture, throughput up to 160 MB/s is required. In contrast, for RR-2week, the lowest throughput configuration (i.e., 5MB/s) is enough to sustain the write rates in all tested workloads. Figure 17 shows the response time of the baseline and RR-2week under various LDPC throughput configurations. The response time reported in this figure is the average of the response time normalized to that with unlimited LDPC throughput:

$$\frac{1}{N}\sum_{i=1}^{N}\left(\frac{ResponseTime_i}{IdealResponseTime_i}\right) \tag{19}$$

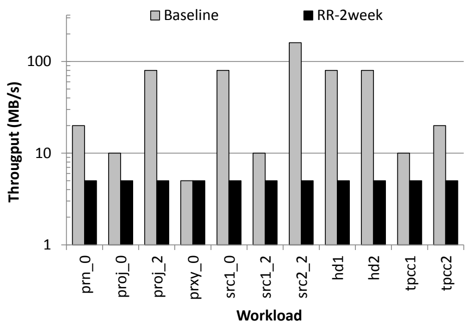


Figure 16: Minimum required LDPC throughput configurations without causing I/O queue saturation

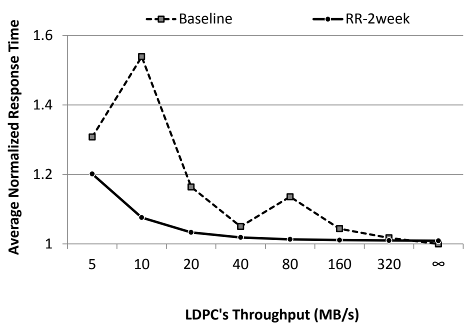


Figure 17: Average normalized response time given various LDPC throughput configurations

where N is the number of workloads which do not incur I/O queue saturation given specific LDPC throughput. In the figure, the curve of the baseline presents a zigzag appearance between 5 MB/s to 80 MB/s because several traces are excluded due to the saturation in the I/O queue. This may inflate the performance of the baseline. Even so, we see RR-2week outperforms the baseline significantly with the same LDPC throughput configuration. For example, with 10MB/s throughput, RR-2week performs 43% better than the baseline. Only when the LDPC throughput approaches infinite does RR-2week perform a bit worse than the baseline due to reprogramming overhead. We can also see that with a 20 MB/s LDPC, RR-2week already approaches the performance of unlimited LDPC throughput, while the baseline requires 160 MB/s to achieve the similar level. Because hardware parallelization is one basic approach to increase the throughput of a LDPC encoder [24], in this point of view, retention relaxation can reduce the hardware cost of LDPC encoders by 8×.

7 Related Work

Access frequencies are usually considered in storage optimization. Chiang *et al.* [12] propose to cluster data with similar write frequencies together to increase SSDs' cleaning efficiency. Pritchett and Thottethodi [33] observe the skewness of disk access frequencies in datacenters and propose novel ensemble-level SSD-based disk caches. In contrast, we focus on the time interval between two successive writes to the same address which defines the data retention requirement.

Several device-aware optimizations for NAND Flash-based SSDs were proposed recently. Grupp *et al.* [17] exploit the variation in page write speed in MLC NAND Flash to improve SSDs' responsiveness. Tanakamaru *et al.* [40] propose wear-out-aware ECC schemes to improve the ECC capability. Xie *et al.* [42] improve write speed through compressing user data and employing stronger ECC codes. Pan *et al.* [31] improve write speed and defect tolerance using wear-out-aware policies. Our work considers the retention requirements of real workloads and relaxes NAND Flash's data retention to optimize SSDs, which is orthogonal to the above device-aware optimization.

Smullen *et al.* [36] and Sun *et al.* [38] improve energy and latency of STTRAM-based CPU caches through redesigning STTRAM cells with relaxed non-volatility. In contrast, we focus on NAND Flash memories used in storage systems.

8 Conclusions

We present the first work on optimizing SSDs via relaxing NAND Flash's data retention capability. We develop a NAND Flash model to evaluate the benefits if NAND Flash's original multi-year data retention can be reduced. We also demonstrate that in real systems, write requests usually require days or even shorter retention times. To optimize the write speed and ECCs' cost and performance, we design SSD systems which handle host writes with shortened retention time while handling background writes as usual and present corresponding retention tracking schemes to guarantee that no data loss happens due to a shortage of retention capability. Simulation results show that the proposed SSDs achieve 1.8–5.7× write response time speedup. We also show that for future SSDs, retention relaxation can bring both performance and cost benefits to the ECC architecture. We leave simultaneously optimizing write speed and ECCs as our future work.

Acknowledgements

We would like to thank our program committee shepherd Bill Bolosky and the anonymous reviewers for their insightful comments and constructive suggestions. This research is supported in part by research grants from ROC National Science Council 100-2219-E-002-027, 100-2219-E-002-030, 100-2220-E-002-015; Excellent Research Projects of National Taiwan University 10R80919-2; Macronix International Co., Ltd. 99-S-C25; and Etron Technology Inc., 10R70152, 10R70156.

References

[1] Fusion-IO ioDrive Duo. http://www.fusionio.com/products/iodrive-duo/.

[2] Hadoop. http://hadoop.apache.org/.

[3] Hammerora. http://hammerora.sourceforge.net/.

[4] NAND Flash support table, July 2011. http://www.linux-mtd.infradead.org/nand-data/nanddata.html.

[5] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *Proc. 2008 USENIX Annual Technical Conference (USENIX '08)*, June 2008.

[6] D. G. Andersen and S. Swanson. Rethinking Flash in the data center. *IEEE Micro*, 30:52–54, July 2010.

[7] F. Arai, T. Maruyama, and R. Shirota. Extended data retention process technology for highly reliable Flash EEPROMs of 10^6 to 10^7 W/E cycles. In *Proc. 1998 IEEE International Reliability Physics Symposium (IRPS '98)*, April 1998.

[8] R. C. Bose and D. K. R. Chaudhuri. On a class of error correcting binary group codes. *Information and Control*, 3(1):68–79, March 1960.

[9] J. Brewer and M. Gill. *Nonvolatile Memory Technologies with Emphasis on Flash: A Comprehensive Guide to Understanding and Using Flash Memory Devices*. Wiley-IEEE Press, 2008.

[10] J. S. Bucy, J. Schindler, S. W. Schlosser, and G. R. Ganger. The DiskSim simulation environment version 4.0 reference manual reference manual (CMU-PDL-08-101). Technical report, Parallel Data Laboratory, 2008.

[11] A. M. Caulfield, L. M. Grupp, and S. Swanson. Gordon: using Flash memory to build fast, power-efficient clusters for data-intensive applications. In *Proc.* 14^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*, March 2009.

[12] M.-L. Chiang, P. C. H. Lee, and R.-C. Chang. Using data clustering to improve cleaning performance for Flash memory. *Softw. Pract. Exper.*, 29:267–290, March 1999.

[13] G. Dong, N. Xie, and T. Zhang. On the use of soft-decision error-correction codes in NAND Flash memory. *IEEE Trans. Circuits Syst. Regul. Pap.*, 58(2):429 –439, February 2011.

[14] D. Floyer. Flash pricing trends disrupt storage. Technical report, May 2010.

[15] R. Gallager. Low-density parity-check codes. *IRE Trans. Inf. Theory*, 8(1):21 –28, January 1962.

[16] J. Gray. Tape is dead. Disk is tape. Flash is disk. RAM locality is king. Presented at the CIDR Gong Show, January 2007.

[17] 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 *Proc.* 42^{nd} *IEEE/ACM International Symposium on Microarchitecture (MICRO '09)*, December 2009.

[18] A. Hocquenghem. Codes correcteurs d'Erreurs. *Chiffres (paris)*, 2:147–156, September 1959.

[19] JEDEC Solid State Technology Association. *Stress-Test-Driven Qualification of Integrated Circuits, JESD47G.01*, April 2010. http://www.jedec.org/.

[20] JEDEC Solid State Technology Association. *Failure Mechanisms and Models for Semiconductor Devices, JEP122G*, October 2011. http://www.jedec.org/.

[21] S. Kopparthi and D. Gruenbacher. Implementation of a flexible encoder for structured low-density parity-check codes. In *Proc. 2007 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM'07)*, Augeust 2007.

[22] S. Li and T. Zhang. Approaching the information theoretical bound of multi-level NAND Flash memory storage efficiency. In *Proc. 2009 IEEE International Memory Workshop (IMW '09)*, May 2009.

[23] S. Li and T. Zhang. Improving multi-level NAND Flash memory storage reliability using concatenated BCH-TCM coding. *IEEE Trans. Very Large Scale Integr. VLSI Syst.*, 18(10):1412 –1420, October 2010.

[24] Z. Li, L. Chen, L. Zeng, S. Lin, and W. Fong. Efficient encoding of quasi-cyclic low-density parity-check codes. *IEEE Trans. Commun.*, 54(1):71 – 81, January 2006.

[25] C.-Y. Lin, C.-C. Wei, and M.-K. Ku. Efficient encoding for dual-diagonal structured LDPC codes based on parity bit prediction and correction. In *Proc. 2008 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS '08)*, December 2008.

[26] R. Micheloni, L. Crippa, and A. Marelli. *Inside NAND Flash Memories*. Springer, 2010.

[27] N. Mielke, T. Marquart, N. Wu, J. Kessenich, H. Belgal, E. Schares, F. Trivedi, E. Goodness, and L. Nevill. Bit error rate in NAND Flash memories. In *Proc. 2008 IEEE International Reliability Physics Symposium (IRPS '08)*, May 2008.

[28] R. Motwani, Z. Kwok, and S. Nelson. Low density parity check (LDPC) codes and the need for stronger ECC. Presented at the Flash Memory Summit, August 2011.

[29] D. Narayanan, A. Donnelly, and A. Rowstron. Write off-loading: Practical power management for enterprise storage. *Trans. Storage*, 4:10:1–10:23, November 2008.

[30] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron. Migrating server storage to SSDs: analysis of tradeoffs. In *Proc.* 4^{th} *ACM European Conference on Computer Systems (EuroSys '09)*, April 2009.

[31] Y. Pan, G. Dong, and T. Zhang. Exploiting memory device wear-out dynamics to improve NAND Flash memory system performance. In *Proc.* 9^{th} *USENIX Conference on File and Stroage Technologies (FAST '11)*, February 2011.

[32] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *Proc.* 5^{th} *USENIX Conference on File and Stroage Technologies (FAST '07)*, February 2007.

[33] T. Pritchett and M. Thottethodi. SieveStore: a highly-selective, ensemble-level disk cache for cost-performance. In *Proc.* 37^{th} *annual International Symposium on Computer Architecture (ISCA '10)*, June 2010.

[34] Samsung Electronics. *K9F8G08UXM Flash memory datasheet*, March 2007.

[35] J.-Y. Shin, Z.-L. Xia, N.-Y. Xu, R. Gao, X.-F. Cai, S. Maeng, and F.-H. Hsu. FTL design exploration in reconfigurable high-performance SSD for server applications. In *Proc.* 23^{rd} *International Conference on Supercomputing (ICS '09)*, 2009.

[36] C. Smullen, V. Mohan, A. Nigam, S. Gurumurthi, and M. Stan. Relaxing non-volatility for fast and energy-efficient STT-RAM caches. In *Proc.* 17^{th} *IEEE International Symposium on High Performance Computer Architecture (HPCA '11)*, February 2011.

[37] K.-D. Suh, B.-H. Suh, Y.-H. Lim, J.-K. Kim, Y.-J. Choi, Y.-N. Koh, S.-S. Lee, S.-C. Kwon, B.-S. Choi, J.-S. Yum, J.-H. Choi, J.-R. Kim, and H.-K. Lim. A 3.3 V 32 Mb NAND Flash memory with incremental step pulse programming scheme. *IEEE J. Solid-State Circuits*, 30(11):1149 –1156, November 1995.

[38] Z. Sun, X. Bi, H. L. nad Weng-Fai Wong, Z. liang Ong, X. Zhu, and W. Wu. Multi retention level STT-RAM cache designs with a dynamic refresh scheme. In *Proc.* 44^{th} *IEEE/ACM International Symposium on Microarchitecture (MICRO '11)*, December 2011.

[39] S. Swanson. Flash memory overview. cse240a: Graduate Computer Architecture, University of California, San Diego, November 2011. http://cseweb.ucsd.edu/classes/fa11/cse240A-a/Slides1/18-FlashOverview.pdf.

[40] S. Tanakamaru, A. Esumi, M. Ito, K. Li, and K. Takeuchi. Post-manufacturing, 17-times acceptable raw bit error rate enhancement, dynamic codeword transition ECC scheme for highly reliable solid-state drives, SSDs. In *Proc. 2010 IEEE International Memory Workshop (IMW '10)*, May 2010.

[41] S. Tanakamaru, C. Hung, A. Esumi, M. Ito, K. Li, and K. Takeuchi. 95%-lower-BER 43%-lower-power intelligent solid-state drive (SSD) with asymmetric coding and stripe pattern elimination algorithm. In *Proc. 2011 IEEE International Solid-State Circuits Conference (ISSCC '11)*, February 2011.

[42] N. Xie, G. Dong, and T. Zhang. Using lossless data compression in data storage systems: Not for saving space. *IEEE Trans. Comput.*, 60:335–345, 2011.

[43] N. Xie, W. Xu, T. Zhang, E. Haratsch, and J. Moon. Concatenated low-density parity-check and BCH coding system for magnetic recording read channel with 4 kB sector format. *IEEE Trans. Magn.*, 44(12):4784 –4789, December 2008.

[44] H. Yasotharan and A. Carusone. A flexible hardware encoder for systematic low-density parity-check codes. In *Proc.* 52^{nd} *IEEE International Midwest Symposium on Circuits and Systems (MWSCAS '09)*, Auguest 2009.

SFS: Random Write Considered Harmful in Solid State Drives

Changwoo Min[a], Kangnyeon Kim[b], Hyunjin Cho[c], Sang-Won Lee[d], Young Ik Eom[e]
[abde]*Sungkyunkwan University, Korea*
[ac]*Samsung Electronics, Korea*
{multics69[a], kangnuni[b],wonlee[d],yieom[e]}@ece.skku.ac.kr, hj1120.cho[c]@samsung.com

Abstract

Over the last decade we have witnessed the relentless technological improvement in flash-based solid-state drives (SSDs) and they have many advantages over hard disk drives (HDDs) as a secondary storage such as performance and power consumption. However, the random write performance in SSDs still remains as a concern. Even in modern SSDs, the disparity between random and sequential write bandwidth is more than tenfold. Moreover, random writes can shorten the limited lifespan of SSDs because they incur more NAND block erases per write.

In order to overcome these problems due to random writes, in this paper, we propose a new file system for SSDs, SFS. First, SFS exploits the maximum write bandwidth of SSD by taking a log-structured approach. SFS transforms all random writes at file system level to sequential ones at SSD level. Second, SFS takes a new data grouping strategy *on writing*, instead of the existing data separation strategy *on segment cleaning*. It puts the data blocks with similar update likelihood into the same segment. This minimizes the inevitable segment cleaning overhead in any log-structured file system by allowing the segments to form a sharp bimodal distribution of segment utilization.

We have implemented a prototype SFS by modifying Linux-based NILFS2 and compared it with three state-of-the-art file systems using several realistic workloads. SFS outperforms the traditional LFS by up to 2.5 times in terms of throughput. Additionally, in comparison to modern file systems such as ext4 and btrfs, it drastically reduces the block erase count inside the SSD by up to 7.5 times.

1 Introduction

NAND flash memory based SSDs have been revolutionizing the storage system. An SSD is a purely electronic device with no mechanical parts, and thus can provide lower access latencies, lower power consumption, lack of noise, shock resistance, and potentially uniform random access speed. However, there remain two serious problems limiting wider deployment of SSDs: limited lifespan and relatively poor random write performance. The limited lifespan of SSDs remains a critical concern in reliability-sensitive environments, such as data centers [5]. Even worse, the ever-increased bit density for higher capacity in NAND flash memory chips has resulted in a sharp drop in the number of program/erase cycles from 10K to 5K for the last two years [4]. Meanwhile, previous work [12, 9] shows that random writes can cause internal fragmentation of SSDs and thus lead to performance degradation by an order of magnitude. In contrast to HDDs, the performance degradation in SSDs caused by the fragmentation lasts for a while after random writes are stopped. The reason for this is that random writes cause the data pages in NAND flash blocks to be copied elsewhere and erased. Therefore, the lifespan of an SSD can be drastically reduced by random writes.

Not surprisingly, researchers have devoted much effort to resolving these problems. Most of work has been focused on a *flash translation layer* (FTL) – an SSD firmware emulating an HDD by hiding the complexity of NAND flash memory. Some studies [24, 14] improved random write performance by providing more efficient logical to physical address mapping. Meanwhile, other studies [22, 14] propose a separation of hot/cold data to improve random write performance. However, such under-the-hood optimizations are purely based on logical block addresses (LBA) requested by a file system so that they would become much less effective for the no-overwrite file systems [16, 48, 10] in which every write to the same file block is always redirected to a new LBA. There are other attempts to improve random write performance especially for database systems [23, 39]. Each attempt proposes a new database storage scheme, taking into account the performance characteristics of SSDs. However, despite the fact that these flash-conscious techniques are quite effective in specific applications, they cannot provide the benefit of such optimization to general applications.

In this paper, we propose a novel file system, SFS, that can improve random write performance and extend the lifetime of SSDs. Our work is motivated by LFS [32], which writes all modifications to disk sequentially in a log-like structure. In LFS, the segment cleaning overhead can severely degrade performance [35, 36] and shorten the lifespan of an SSD. This is because quite a high number of pages need to be copied to secure a large empty chunk for a sequential write at every segment cleaning. In designing SFS, we investigate how to take advantage of performance characteristics of SSD and I/O workload skewness to reduce the segment cleaning overhead.

This paper makes the following specific contributions:

- We introduce the design principles for SSD-based file systems. The file system should exploit the performance characteristics of SSD and directly utilize file block level statistics. In fact, the architectural differences between SSD and HDD results in different performance characteristics for each system. One interesting example is that, in SSD, the additional overhead of random write disappears only when the unit size of random write requests becomes a multiple of a certain size. To this end, we take log-structured approach with a carefully selected segment size.
- To reduce the segment cleaning overhead in the log-structured approach, we propose an eager *on writing* data grouping scheme that classifies file blocks according to their update likelihood and writes those with similar update likelihoods into the same segment. The effectiveness of data grouping is determined by proper selection of the grouping criteria. For this, we propose an *iterative segment quantization* algorithm to determine the grouping criteria, while considering disk-wide hotness distribution. We also propose *cost-hotness policy* for victim segment selection. Our eager data grouping will colocate frequently updated blocks in the same segments; thus most blocks in those segments are expected to become rapidly invalid. Consequently, the segment cleaner can easily find a victim segment with few live blocks and thus can minimize the overhead of copying the live blocks.
- Using a number of realistic and synthetic workloads, we show that SFS significantly outperforms LFS and state-of-the-art file systems such as ext4 and btrfs. We also show that SFS can extend the lifespan of an SSD by drastically reducing the number of NAND flash block erases. In particular, while the random write performance of the existing file systems is highly dependent on the random write performance of SSD, SFS can achieve nearly maximum sequential write bandwidth of SSD for random writes at the file system level. Therefore, SFS can provide high performance even on mid-range or low-end SSDs as long as their sequential write performance is comparable to high-end SSDs.

The rest of this paper is organized as follows. Section 2 overviews the characteristics of SSD and I/O workloads. Section 3 describes the design of SFS in detail, and Section 4 shows the extensive evaluation of SFS. Related work is described in Section 5. Finally, in Section 6, we conclude the paper.

2 Background

2.1 Flash Memory and SSD Internals

NAND flash memory is the basic building block of SSDs. *Read* and *write* operations are performed at the granularity of a *page* (e.g. 2 KB or 4 KB), and the *erase* operation is performed at the granularity of a *block* (composed of 64 – 128 pages). NAND flash memory differs from HDDs in several aspects: (1) asymmetric speed of read and write operations, (2) no in-place overwrite – the whole block must be erased before overwriting any page in that block, and (3) limited program/erase cycles – a single-level cell (SLC) has roughly 100K erase cycles and a typical multi-level cell (MLC) has roughly 10K erase cycles.

A typical SSD is composed of host interface logic (SATA, USB, and PCI Express), an array of NAND flash memories, and an SSD controller. A *flash translation layer* (FTL) run by an SSD controller emulates an HDD by exposing a linear array of *logical block addresses* (LBAs) to the host. To hide the unique characteristics of flash memory, it carries out three main functions: (1) managing a *mapping table* from LBAs to physical block addresses (PBAs), (2) performing *garbage collection* to recycle invalidated physical pages, and (3) *wear-leveling* to wear out flash blocks evenly in order to extend the SSD's lifespan. Agrawal et al. [2] comprehensively describe the broad design space and tradeoffs of SSD.

Much research has been carried out on FTL to improve performance and extend the lifetime [18, 24, 22, 14]. In a *block-level FTL* scheme, a logical block number is translated to a physical block number and the logical page offset within a block is fixed. Since the mapping in this instance is coarse-grained, the mapping table is small enough to be kept in memory entirely. Unfortunately, this results in a higher garbage collection overhead. In contrast, since a *page-level FTL* scheme manages a fine-grained page-level mapping table, it results in a lower garbage collection overhead. However, such fine-grained mapping requires a large mapping table on RAM. To overcome such technical difficulties, *hybrid FTL* schemes [18, 24, 22] extend the block-level FTL. These schemes logically partition flash blocks into *data blocks* and *log blocks*. The majority of data blocks are mapped using block level mapping to reduce the required RAM size and log blocks are mapped using page-level mapping to reduce the garbage collection overhead. Similarly, DFTL [14] extends the page-level mapping by

	SSD-H	SSD-M	SSD-L
Manufacturer	Intel	Samsung	Transcend
Model	X25-E	S470	JetFlash 700
Capacity	32 GB	64 GB	32 GB
Interface	SATA	SATA	USB 3.0
Flash Memory	SLC	MLC	MLC
Max Sequential Reads (MB/s)	216.9	212.6	69.1
Random 4 KB Reads (MB/s)	13.8	10.6	5.3
Max Sequential Writes (MB/s)	170	87	38
Random 4 KB Writes (MB/s)	5.3	0.6	0.002
Price ($/GB)	14	2.3	1.4

Table 1: Specification data of the flash devices. List price is as of September 2011.

selectively caching page-level mapping table entries on RAM.

2.2 Imbalance between Random and Sequential Write Performance in SSDs

Most SSDs are built on an array of NAND flash memories, which are connected to the SSD controller via multiple channels. To exploit this inherent parallelism for better I/O bandwidth, SSDs perform read or write operations as a unit of a *clustered page* [19] that is composed of physical pages striped over multiple NAND flash memories. If the request size is not a multiple of the clustered page size, extra read or write operations are performed in the SSD and the performance is degraded. Thus, the clustered page size is critical in I/O performance.

Write performance in SSDs is highly workload dependent and is eventually limited by the garbage collection performance of FTL. Previous work [12, 9, 39, 37, 38] has reported that random write performance drops by more than an order of magnitude after extensive random updates and returns to the initial high performance only after extensive sequential writes. The reason for this is that random writes increase the garbage collection overhead in FTL. In a hybrid FTL, random writes increase the associativity between log blocks and data blocks, and incur the costly *full merge* [24]. In page-level FTL, as it tends to fragment blocks evenly, garbage collection has large copying overhead.

In order to improve garbage collection performance, SSD combines several blocks striped over multiple NAND flash memories into a *clustered block* [19]. The purpose of this is to erase multiple physical blocks in parallel. If all write requests are aligned in multiples of the clustered block size and their sizes are also multiples of the clustered block size, random write updates and invalidates a clustered block as a whole. Therefore, in hybrid FTL, a *switch merge* [24] with the lowest overhead occurs. Similarly, in page-level FTL, empty blocks with no live pages are selected as victims for garbage collection. The result of this is that random write performance converges with sequential write performance. To verify this, we measured sequential write and random write throughput on three different SSDs in Table 1, ranging from a high-end SLC SSD (SSD-H) to a low-end USB memory stick (SSD-L). To determine sustained write performance, dummy data equal to twice the device's capacity is first written for aging, and the throughput of subsequent writing for 8GB is measured. Figure 1 shows that random write performance catches up with sequential write performance when the request size is 16 MB or 32 MB. These unique performance characteristics motivate the second design principle of SFS: write bandwidth maximization by sequential writes to SSD.

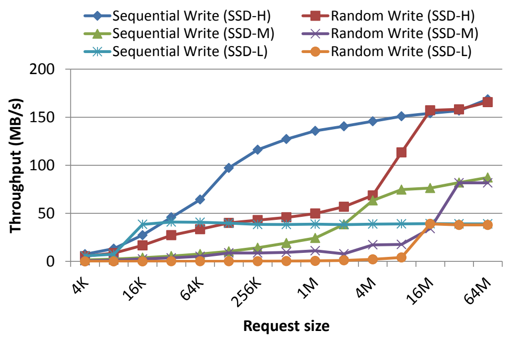


Figure 1: Sequential vs. random write throughput.

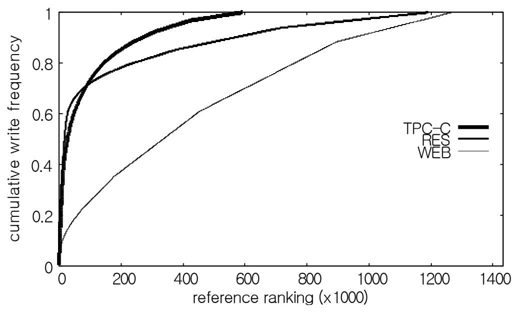


Figure 2: Cumulative write frequency distribution.

2.3 Skewness in I/O Workloads

Many researchers have pointed out that I/O workloads have non-uniform access frequency distribution [34, 31, 23, 6, 3, 33, 11]. A disk-level trace of personal workstations at Hewlett Packard laboratories exhibits a high locality of references in that 90% of the writes go to the 1% of blocks [34]. Roselli et al. [31] analyzed file system traces collected from four different groups of machines: an instructional laboratory, a set of computers used for research, a single web server, and a set of PCs running Windows NT. They found that files tend to be either read-mostly or write-mostly and the writes show substantial locality. Lee and Moon [23] showed that the update frequency of TPC-C workloads is highly skewed, in that 29% writes go to 1.6% of pages.

Bhadkamkar et al. [6] collected and investigated I/O traces of office and developer desktop workloads, a version control server, and a web server. Their analysis confirms that the top 20% most frequently accessed blocks contribute to a substantially large (45-66%) percentage of total access. Moreover, high and low frequency blocks are spread over the entire disk area in most cases. Figure 2 depicts the cumulative write frequency distribution of three real workloads: an IO trace collected by ourselves while running TPC-C [40] using Oracle DBMS (TPC-C), a research group trace (RES), and a web sever trace equipped with Postgres DBMS (WEB) collected by Roselli et al [31]. This observation motivates the third design principle of SFS: block grouping according to write frequency.

3 Design of SFS

SFS is motivated by a simple question: *How can we utilize the performance characteristics of SSD and the skewness of the I/O workload in designing an SSD-based file system?* In this section, we describe the rationale behind the design decisions in SFS, its system architecture, and several key techniques including hotness measure, segment quantization, segment writing, segment cleaning and victim selection policy, and crash recovery.

3.1 SFS: Design for SSD-based File Systems of the 2010s

Historically, existing file systems and modern SSDs have evolved separately without consideration of each other. With the exception of the recently introduced TRIM command, the two layers communicate with each other through simple read and write operations using only LBA information. For this reason, there are many impedance mismatches between the two layers, thus hindering the optimal performance when both layers are simply used together. In this section, we explain three design principles of SFS. First, we identify general performance problems when the existing file systems are running on modern SSDs and suggest that a file system should exploit the file block semantics directly. Second, we propose to take a log-structured approach based on the observation that the random write bandwidth is much slower than the sequential one. Third, we criticize that the existing *lazy* data grouping in LFS during segment cleaning fails to fully utilize the skewness in write patterns and argue that an *eager* data grouping is necessary to achieve sharper bimodality in segments. In followings we will describe each principle in detail.

File block level statistics – Beyond LBA: The existing file systems perform suboptimally when running on top of SSDs with current FTL technology. This suboptimal performance can be attributed to poor random write performance in SSDs. One of the basic functionalities of file systems is to allocate an LBA for a file block. With regard to this LBA allocation, there have been two general policies in file system community: *in-place-update* and *no-overwrite*. The in-place-update file systems such as FAT32 [27] and ext4 [25] always overwrite a dirty file block to the same LBA so that the same LBA ever corresponds to a file block unless the file frees the block. This *unwritten assumption* in file systems, together with the LBA level interface between file systems and storage devices, allows the underlying FTL mechanism in SSDs to exploit the overwrites against the same LBA address. In fact, most FTL research [24, 22, 13, 14] has focused on improving the random write performance based on the LBA level write patterns. Despite the relentless improvement in FTL technology, the random write bandwidth in modern SSDs, as presented in Section 2.2, still lags far behind the sequential one.

Meanwhile, several no-overwrite file systems have been implemented, such as btrfs [10], ZFS [48], and WAFL [16], where dirty file blocks are written to new LBAs. These systems are known to improve scalability, reliability, and manageability [29]. In those systems, however, because the unwritten assumption between file blocks and their corresponding LBAs is broken, the FTL receives new LBA write request for every update of a file block and thus cannot exploit any file level hotness semantics for random write optimization.

In summary, the LBA-based interface between the *no-overwrite* file systems and storage devices does not allow the file blocks' hotness semantic to flow down to the storage layer. In addition, the relatively poor random write performance in SSDs is the source of suboptimal performance in the *in-place-update* file systems. Consequently, we suggest that file systems should directly exploit the hotness statistics at the *file block level*. This allows for optimization of the file system performance regardless of whether the unwritten assumption holds and how the underlying SSDs perform on random writes.

Write bandwidth maximization by sequentialized writes to SSD: In Section 2.2, we show that the random write throughput becomes equal to the sequential write throughput only when the request size is a multiple of the clustered block size. To exploit such performance characteristics, SFS takes a log-structured approach that turns random writes at the file level into sequential writes at the LBA level. Moreover, in order to utilize nearly 100% of the raw SSD bandwidth, the segment size is set to a multiple of the clustered block size. The result is that the performance of SFS will be limited by the maximum sequential write performance regardless of random write performance.

Eager *on writing* data grouping for better bimodal segmentation: When there are not enough free segments, a segment cleaner copies the live blocks from vic-

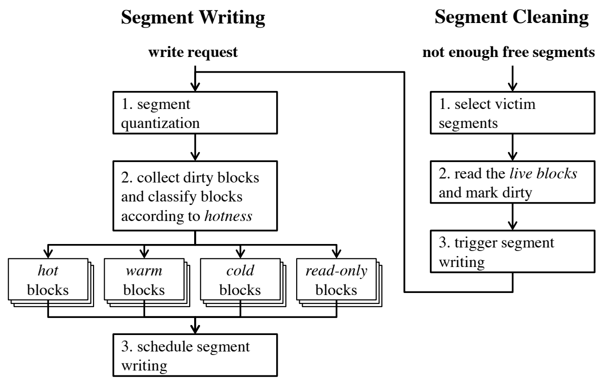

Segment Writing
write request
1. segment quantization
2. collect dirty blocks and classify blocks according to *hotness*
hot blocks
warm blocks
cold blocks
read-only blocks
3. schedule segment writing
Segment Cleaning
not enough free segments
1. select victim segments
2. read the *live blocks* and mark dirty
3. trigger segment writing

Figure 3: Overview of writing process and segment cleaning in SFS.

tim segments in order to secure free segments. Since segment cleaning includes reads and writes of live blocks, it is the main source of overhead in any log-structured file system. Segment cleaning cost becomes especially high when cold data are mixed with hot data in the same segment. Since cold data are updated less frequently, they are highly likely to remain live at the segment cleaning and thus be migrated to new segments. If hot data and cold data are grouped into different segments, most blocks in the hot segment will be quickly invalidated, while most blocks in the cold segment will remain live. As a result, the segment utilization distribution becomes bimodal: most of the segments are almost either full or empty of live blocks. The cleaning overhead is drastically reduced, because the segment cleaner can almost always work with nearly empty segments. To form a bimodal distribution, LFS uses a cost-benefit policy [32] that prefers cold segments over hot segments. However, LFS writes data regardless of hot/cold and then tries to separate data lazily *on segment cleaning*. If we can categorize hot/cold data when it is first written, there is much room for improvement.

In SFS, we classify data *on writing* based on file block level statistics as well as segment cleaning. In such early data grouping, since segments are already composed of homogeneous data with similar update likelihood, we can significantly reduce segment cleaning overhead. This is particularly effective because I/O skewness is common in real world workloads, as shown in Section 2.3.

3.2 SFS Architecture

SFS has four core operations: segment writing, segment cleaning, reading, and crash recovery. Segment writing and segment cleaning are particularly crucial for performance optimization in SFS, as depicted in Figure 3. Because the read operation in SFS is same as that of existing log-structured file systems, we will not cover its detail in this paper.

As a measure for representing the future update likelihood of data in SFS, we define *hotness* for file block, file, and segment, respectively. As the hotness is higher, the data is expected to be updated sooner. The first step of segment writing in SFS is to determine the hotness criteria for block grouping. This is, in turn, determined by segment quantization that quantizes a range of hotness values into a single hotness value for a group. For the sake of brevity, it is assumed throughout this paper that there are four segment groups: hot, warm, cold, and read-only groups. The second step of segment writing is to calculate the block hotness for each block and assign them to the nearest quantized group by comparing the block hotness and the group hotness. At this point, those blocks with similar hotness levels should belong to the same group (i.e. their future update likelihood is similar). As the final step of segment writing, SFS always fills a segment with blocks belonging to the same group. If the number of blocks in a group is not enough to fill a segment, the segment writing of the group is deferred until the segment is filled. This eager grouping of file blocks according to the hotness measure serves to colocate blocks with similar update likelihoods in the same segment. Therefore, segment writing in SFS is very effective at achieving sharper bimodality in segment utilization distribution.

Segment cleaning in SFS consists of three steps: select victim segments, read the live blocks in victim segments into the page cache and mark the live blocks as dirty, and trigger the writing process. The writing process treats the live blocks from victim segments the same as normal blocks; each live block is classified into a specific quantized group according to its hotness. After all the live blocks are read into the page cache, the victim segments are then marked as free so that they can be reused for writing. For better victim segment selection, *cost-hotness policy* is introduced, which takes into account both the number of live blocks in segment (i.e. cost) and the segment hotness.

In the following sections, we will explain each component of SFS in detail: how to measure hotness (§ 3.3), segment quantization (§ 3.4), segment writing (§ 3.5), segment cleaning (§ 3.6), and crash recovery (§ 3.7).

3.3 Measuring Hotness

In SFS, *hotness* is used as a measure of how likely the data is to be updated. Hotness is defined for file block, file, and segment, respectively. Although it is difficult to estimate data hotness without prior knowledge of future access pattern, SFS exploits both the skewness and the temporal locality in the I/O workload so as to estimate the update likelihood of data. From the skewness observed in many workloads, frequently updated data tends to be updated quickly. Moreover, because of the temporal locality in references, the recently updated data is likely to be changed quickly. Thus, using the skewness and the temporal locality, *hotness* is defined as $\frac{\text{write count}}{\text{age}}$. Each hotness of file block, file, and segment is specifically defined as follows.

First, *block hotness* H_b is defined by age and write count of a block as follows:

$$H_b = \begin{cases} \frac{W_b}{T - T_b^m} & \text{if } W_b > 0, \\ H_f & \text{otherwise.} \end{cases}$$

where T is the current time, T_b^m is the last modified time of the block, and W_b is the total number of writes on the block since the block was created. If a block is newly created ($W_b = 0$), H_b is defined as the hotness of the file that the block belongs to.

Next, *file hotness* H_f is used to estimate the hotness of a newly created block. It is defined by age and write count of a file as follows:

$$H_f = \frac{W_f}{T - T_f^m}$$

where T_f^m is the last modified time of the file, and W_f is the total number of block updates since the file was created.

Finally, *segment hotness* represents how likely a segment is to be updated. Since a segment is a collection of blocks, it is reasonable to derive its hotness from the hotness of live blocks contained within. That is, as the hotness of live blocks in a segment is higher, the segment hotness also becomes higher. Therefore, we define hotness of a segment H_s as the average hotness of the live blocks in the segment. However, it is expensive to calculate H_s because the liveness of all blocks in a segment must be tested. To determine H_s for all segments in a disk, the liveness of all blocks in the disk must be checked. To alleviate this cost, we approximately calculate the average hotness of live blocks in a segment as follows:

$$\begin{aligned} H_s &= \frac{1}{N}\sum_i H_{b_i} \\ &\approx \frac{\text{mean of write count of live blocks}}{\text{mean of age of live blocks}} \\ &= \frac{\sum_i W_{b_i}}{N \cdot T - \sum_i T_{b_i}^m} \end{aligned}$$

where N is the number of live blocks in a segment, H_{b_i}, $T_{b_i}^m$, and W_{b_i} are block hotness, last modified time, and write count of i-th live block, respectively. When a segment is created, SFS stores $\sum_i T_{b_i}^m$ and $\sum_i W_{b_i}$ in the segment usage meta-data file (SUFILE), and updates them by subtracting $T_{b_i}^m$ and W_{b_i} whenever a block is invalidated. Using this approximation, we can incrementally calculate H_s of a segment without checking the liveness of blocks in the segment. We will elaborate on how to manage meta-data for hotness in Section 4.1.

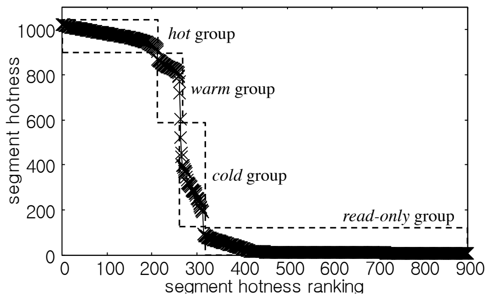


Figure 4: Example of segment quantization.

3.4 Segment Quantization

In order to minimize the overhead of copying the live blocks during segment cleaning, it is crucial for SFS to properly group blocks according to hotness and then to write them in grouped segments. The effectiveness of block grouping is determined by the grouping criteria. In fact, improper criteria may colocate blocks from different groups into the same segment, thus deteriorating the effectiveness of grouping. Ideally, grouping criteria should consider the distribution of all blocks' hotness in the file system, yet in reality this is too costly. Thus, we instead use segment hotness as an approximation of block hotness and devise an algorithm to calculate the criterion, *iterative segment quantization*.

In SFS, *segment quantization* is a process used to partition the hotness range of a file system into k sub-ranges and calculate a quantized value for each sub-range representing a group. There are many alternative ways to quantize hotness. For example, each group can be quantized using *equi-height partitioning* or *equi-width partitioning*. Equi-height partitioning equally divides the whole hotness range into multiple groups and equi-width partitioning makes each group have an equal number of segments. In Figure 4, the segment hotness distribution is computed by measuring the hotness for all segments on the disk after running TPC-C workload under 70% disk utilization. In such a distribution where most segments are not hot, however, both approaches fail to correctly reflect the hotness distribution and the resulting group quantization is suboptimal.

In order to correctly reflect the hotness distribution of segments and to properly quantize them, we propose an *iterative segment quantization* algorithm. Inspired by the data clustering approach in statistics domain [15], our iterative segment quantization partitions segments into k groups and tries to find the centers of natural groups through an iterative refinement approach. A detailed description of the algorithm is as follows:

1. If the number of written segments is less than or equal to k, assign a randomly selected segment hotness to initial value of H_{g_i}, which denotes hotness of the i-th group.
2. Otherwise update H_{g_i} as follows:
   (a) Assign each segment to the group G_i whose hotness is closest to the segment hotness.
   $$G_i = \{H_{s_j} : \|H_{s_j} - H_{g_i}\| \le \|H_{s_j} - H_{g_{i^*}}\| \text{ for all } i^* = 1, \ldots, k\}$$
   (b) Calculate the new means to be the group hotness H_{g_i}.
   $$H_{g_i} = \frac{1}{|G_i|} \sum_{H_{s_j} \in G_i} H_{s_j}$$
3. Repeat Step 2 until H_{g_i} no longer changes or three times at most.

Despite the fact that its computational overhead increases in proportion to the number of segments, the large segment size means that the overhead of the proposed algorithm is reasonable (32 segments for 1 GB disk space given 32 MB segment size). To further reduce the overhead, SFS stores H_{g_i} in meta-data and reloads them at mounting for faster convergence.

3.5 Segment Writing

As illustrated in Figure 3, segment writing in SFS consists of two sequential steps: one to group dirty blocks in the page cache and the other to write the blocks groupwise in segments. Segment writing is invoked in four cases: (a) SFS periodically writes dirty blocks every five seconds, (b) flush daemon forces a reduction in the number of dirty pages in the page cache, (c) segment cleaning occurs, and (d) an *fsync* or *sync* occurs. The first step of segment writing is segment quantization: all H_{g_i} are updated as described in Section 3.4. Next, the block hotness H_b of every dirty block is calculated, and each block is assigned to the group H_{g_i} whose hotness is closest to the block hotness.

To avoid blocks in different groups being colocated in the same segment, SFS completely fills a segment with blocks from the same group. In other words, among all groups, only the groups large enough to completely fill a segment are written. Thus, when the group size, i.e. the number of blocks belonging to a group, is less than the segment size, SFS will defer writing the blocks to the segment until the group size reaches the segment size. However, when an *fsync* or *sync* occurs or SFS initiates a *check-point*, every dirty block including the deferred blocks should be immediately written to segment regardless of the group size. In this case, we take a best-effort approach: at first, writing out blocks groupwise as many as possible, then writing only the remaining blocks regardless of group. In all cases, writing a block accompanies updating relevant meta-data, T_b^m, W_b, T_f^m, W_f, $\sum_i T_{b_i}^m$, and $\sum_i W_{b_i}$, and invalidating the liveness of the overwritten block. Since the writing process continuously reorganizes file blocks according to hotness, it helps to form sharp bimodal distribution of segment utilization, and thus to reduce the segment cleaning overhead. Further, it almost always generates aligned large sequential write requests that are optimal for SSD.

Because the blocks under segment cleaning are handled similarly, their writing can also be deferred if the number of live blocks belonging to a group is not enough to completely fill a segment. As such, there is a danger that the not-yet-written blocks under segment cleaning might be lost if the originating segments of the blocks are already overwritten by new data but a system crash or a sudden power off is encountered. To cope with such data loss, two techniques are introduced. First, SFS manages a free segment list and allocates segments in the *least recently freed (LRF)* order. Second, SFS checks whether writing a normal block could cause a not-yet-written block under segment cleaning to be overwritten. Let S^t denote a newly allocated segment and S^{t+1} denote a segment that will be allocated in next segment allocation. If there are not-yet-written blocks under segment cleaning that originate in S^{t+1}, SFS writes such blocks to S^t regardless of grouping. This guarantees that not-yet-written blocks under segment cleaning are never overwritten before they are written elsewhere. The segment-cleaned blocks are thus never lost, even in a system crash or a sudden power off, because they always have an on-disk copy. The LRF allocation scheme increases the opportunity for a segment-cleaned block to be written by block grouping rather than this scheme. The details of minimizing the overhead in this process are omitted from this paper.

3.6 Segment Cleaning: Cost-hotness policy

In any log-structured file system, the victim selection policy is critical to minimizing the overhead of segment cleaning. There are two well-known segment cleaning policies: *greedy policy* [32] and *cost-benefit policy* [32, 17]. Greedy policy [32] always selects segments with the smallest number of live blocks, hoping to reclaim as much space as possible with the least copying out overhead. However, it does not consider the hotness of data blocks during segment cleaning. In practice, because the cold data tends to remain unchanged for a long time before it becomes invalidated, it would be very beneficial to separate cold data from hot data. To this end, cost-benefit policy [32, 17] prefers cold segments to hot segments when the number of live blocks is equal. Even though it is critical to estimate how long a segment remains unchanged, cost-benefit policy simply uses the last modified time of any block in the segment (i.e. the age of the youngest block) as a simple measure of the segment's update likelihood.

As a natural extension of cost-benefit policy, we introduce *cost-hotness policy*; since hotness in SFS directly represents the update likelihood of segment, we use segment hotness instead of segment age. Thus, SFS chooses a victim among the segments, which maximizes the following formula:

$$\begin{aligned}\text{cost-hotness} &= \frac{\text{free space generated}}{\text{cost} * \text{segment hotness}}\\ &= \frac{(1-U_s)}{2U_sH_s}\end{aligned}$$

where U_s is segment utilization, i.e. the fraction of the live blocks in a segment. The cost of collecting a segment is $2U_s$ (one U_s to read valid blocks and the other U_s to write them back). Although cost-hotness policy needs to access the utilization and the hotness of all segments, it is very efficient because our implementation keeps them in segment usage meta-data file (SUFILE) and meta-data size per segment is quite small (48 bytes long). All segment usage information is very likely to be cached in memory and can be accessed without accessing the disk in most cases. We will describe the detail of meta-data management in Section 4.1.

In SFS, the segment cleaner is invoked when the disk utilization exceeds a *water-mark*. The water-mark for the our experiments is set to 95% of the disk capacity and the segment cleaning is allowed to process up to three segments at once (96 MB given the segment size of 32 MB). The prototype did not implement the idle time cleaning scheme suggested by Blackwell et al. [7], yet this could be seamlessly integrated with SFS.

3.7 Crash Recovery

Upon a system crash or a sudden power off, the in progress write operations may leave the file system inconsistent. This is because dirty data blocks or meta-data blocks in the page cache may not be safely written to the disk. In order to restore such inconsistencies from failures, SFS uses a *check-point* mechanism; on re-mounting after a crash, the file system is rolled back to the last check-point state, and then resumes in a normal manner. A check-point is the state in which all of the file system structures are consistent and complete. In SFS, a check-point is carried out in two phases; first, it writes out all the dirty data and meta-data to the disk, and then updates the superblock in a special fixed location on the disk. The superblock keeps the root address of the meta-data, the position in the last written segment and time-stamp. SFS can guarantee the atomic write of the superblock by alternating between writing it to two separate physical blocks on the disk. During re-mounting, SFS reads both copies of the superblock, compares their time stamps and uses the more recent one.

Frequent check-pointing can minimize data loss from crashes but can hinder normal system performance. Considering this trade-off, SFS performs a check-point in four cases: (a) every thirty seconds after creating a check-point, (b) when more than 20 segments (640 MB given a segment size of 32 MB) are written, (c) when performing *sync* or *fsync* operation, and (d) when the file system is unmounted.

4 Evaluation

4.1 Experimental Systems

Implementation: SFS is implemented based on NILFS2 [28] by retrofitting the in-memory and on-disk meta-data structures to support block grouping and cost-hotness segment cleaning. NILFS2 in the mainline Linux kernel is based on log-structured file system [32] and incorporates advanced features such as b-tree based block management for scalability and continuous snapshot [20] for ease of management.

Implementing SFS requires a significant engineering effort, despite the fact that it is based on the already existing NILFS2. NILFS2 uses b-tree for scalable block mapping and virtual-to-physical block translation in data address translation (DAT) meta-data file to support continuous snapshot. One technical issue of b-tree based block mapping is the excessive meta-data update overhead. If a leaf block in a b-tree is updated, its effect is always propagated up to the root node and all the corresponding virtual-to-physical entries in the DAT are also updated. Consequently, random writes entail a significant amount of meta-data updates — writing 3.2 GB with 4 KB I/O unit generates 3.5 GB of meta-data. To reduce this meta-data update overhead and support the check-point creation policy discussed in Section 3.7, we decided to cut off the continuous snapshot feature. Instead, SFS-specific fields are added to several meta-data structures: superblock, inode file (IFILE), segment usage file (SUFILE), and DAT file. Group hotness H_{g_i} is stored in the superblock and loaded at mounting for the iterative segment quantization. Per file write count W_f and the last modified time T_f^m are stored in the IFILE. The SUFILE contains information for hotness calculation and segment cleaning: U_s, H_s, $\sum_i T_{b_i}^m$ and $\sum_i W_{b_i}$. Per-block write count W_b and the last modified time T_b^m are stored in the DAT entry along with virtual-to-physical mapping. Of these, W_b and T_b^m are the largest, each being eight bytes long. Since the meta-data fields for continuous snapshot in the DAT entry have been removed, the size of the DAT entry in SFS is the same as that of NILFS2 (32 bytes). As a result of these changes, we reduce the runtime overhead of meta-data to 5%–10% for the workloads used in our experiments. In SFS, since a meta-data file is treated the same as a normal file with a special inode number, a meta-data file can also be cached in the page cache for efficient access.

Segment cleaning in NILFS2 is not elaborated to the state-of-the-art in academia. It takes simple *time-stamp policy* [28] that selects the oldest dirty segment as a victim. For SFS, we implemented the cost-hotness policy and segment cleaning triggering policy described in Section 3.6.

In our implementation, we used Linux kernel 2.6.37, and all experiments are performed on a PC using a 2.67 GHz Intel Core i5 quad-core processor with 4 GB of physical memory.

Target SSDs: Currently, the spectrum of SSDs available in the market is very wide in terms of price and performance; flash memory chips, RAM buffers, and hardware controllers all vary greatly. For this paper, we select three state-of-the-art SSDs as shown in Table 1. The high-end SSD is based on SLC flash memory and the rest are based on MLC. Hereafter, these three SSDs are referred to as *SSD-H*, *SSD-M*, and *SSD-L* ranging from high-end to low-end.

Figure 1 shows sequential vs. random write throughput of the three devices. The request sizes of random write whose bandwidth converges to that of sequential write are 16 MB, 32 MB, and 16 MB for SSD-H, SSD-M, and SSD-L, respectively. To fully exploit device performance, the segment size is set to 32 MB for all three devices.

Workloads: To study the impact of SFS on various workloads, we use a mixture of synthetic and real-world workloads. Two real-world file system traces are used in our experiments: OLTP database workload, and desktop workload. For OLTP database workload, the file system level trace is collected while running TPC-C [40]. The database server runs Oracle 11g DBMS and the load server runs Benchmark Factory [30] using TPC-C benchmark scenario. For desktop workload, we used RES from the University of California at Berkeley [31]. *RES* is a research workload collected for 113 days on a system consisting of 13 desktop machines of a research group. In addition, two traces of random writes with different distributions are generated as synthetic workloads: one with Zipfian distribution and the other with uniform random distribution. The uniform random write is the workload that shows the worst case behavior of SFS, since SFS tries to utilize the skewness in workloads during block grouping.

Since our main area of interest is in maximum write performance, write requests in the workloads are replayed as fast as possible in a single thread and throughput is measured at the application level. Native Command Queuing (NCQ) is enabled to maximize the parallelism in the SSD. In order to explore the system behavior on various disk utilizations, we sequentially filled the SSD with enough dummy blocks, which are never updated after creation, until the desired utilization is reached. Since the amount of the data block update is the same for a workload regardless of the disk utilization, the amount of the meta-data update is also the same. Therefore, in our experiment results, we can directly compare performance metrics for a workload regardless of the disk utilization.

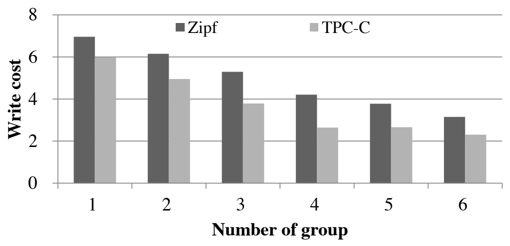


Figure 5: Write cost vs. number of group. Disk utilization is 85%.

Write Cost: To write new data in SFS, a new segment is generated by the segment cleaner. This cleaning process will incur additional read and write operations for the live blocks being segment-cleaned. Therefore, the write cost of data should include the implicit I/O cost of segment cleaning as well as the pure write cost of new data. In this paper, we define the write cost W_c to compare the write cost induced by the segment cleaning. It is defined by three component costs – the write cost of new data W_c^{new}, the read and the write cost of the data being segment-cleaned, R_c^{sc} and W_c^{sc} – as follows:

$$W_c = \frac{W_c^{new} + R_c^{sc} + W_c^{sc}}{W_c^{new}}$$

Each component cost is defined by division of the amount of I/O by throughput. Since the unit of write in SFS is always a large sequential chunk, we choose the maximum sequential write bandwidth in Table 1 for throughputs of W_c^{sc} and W_c^{new}. Meanwhile, since the live blocks being segment-cleaned are assumed to be randomly located in a victim segment, the 4 KB random read bandwidth in Table 1 is selected for the read throughput of R_c^{sc}. Throughout this paper, we measured the amount of I/O while replaying the workload trace and thus calculated the write cost for a workload.

4.2 Effectiveness of SFS Techniques

As discussed in Section 3, the key techniques of SFS are (a) on writing block grouping, (b) iterative segment quantization, and (c) cost-hotness segment cleaning. To

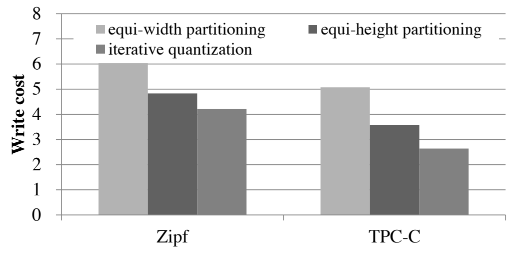


Figure 6: Write costs of quantization schemes. Disk utilization is 85%.

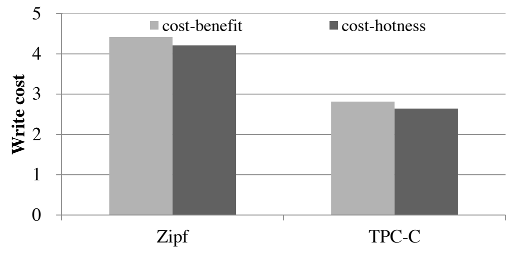


Figure 7: Write cost vs. segment cleaning scheme. Disk utilization is 85%.

examine how each technique contributes to the overall performance, we measured the write costs of Zipf and TPC-C workload under 85% disk utilization on SSD-M.

First, to verify how the block grouping is effective, we measured the write costs by varying the number of groups from one to six. As shown in Figure 5, we can observe that the effect of block grouping is considerable. When the blocks are not grouped (i.e. the number of groups is 1), the write cost is fairly high: 6.96 for Zipf and 5.98 for TPC-C. Even when the number of groups increases to two or three, no significant reduction in write cost is observed. However, when the number of groups reaches four the write costs of Zipf and TPC-C workloads significantly drop to 4.21 and 2.64, respectively. In the case of five or more groups, the write cost reduction is marginal. The additional groups do not help much when there are already enough groups covering hotness distribution, but may in fact increase the write cost. Since more blocks can be deferred due to insufficient blocks in a group, this could result in more blocks being written without grouping when creating a checkpoint.

Next, we compared the write cost of the different segment quantization schemes across four groups. Figure 6 shows that our iterative segment quantization reduces the write costs significantly. The equi-width partitioning scheme has the highest write cost; 143% for Zipf and 192% for TPC-C over the iterative segment quantization. The write costs of the equi-height partitioning scheme are 115% for Zipf and 135% for TPC-C over the iterative segment quantization.

Finally, to verify how cost-hotness policy affects performance, we compared the write cost of cost-hotness policy and cost-benefit policy with the iterative segment quantization for four groups. As shown in Figure 7, cost-hotness policy can reduce the write cost by approximately 7% over for both TPC-C and Zipf workload.

4.3 Performance Evaluation

4.3.1 Write Cost and Throughput

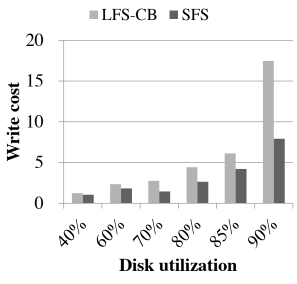
To show how SFS and LFS perform against various workloads with different write patterns, we measured their write costs and throughput for two synthetic workloads and two real workloads, and presented the performance results in Figure 8 and 9. For LFS, we implemented the cost-benefit cleaning policy in our code base (hereafter LFS-CB). Since throughput is measured at the application level, it includes the effects of the page cache and thus can exceed the maximum throughput of each device. Due to space constraints, only the experiments on SSD-M are shown here. The performance of SFS on different devices is shown in Section 4.3.3.

First, let us explain how much SFS can improve the write cost. It is clear from Figure 8 that SFS significantly reduces the write cost compared to LFS-CB. In particular, the relative write cost improvement of SFS over LFS-CB gets higher as disk utilization increases. Since there is not enough time for the segment cleaner to reorganize blocks under high disk utilization, our *on writing* data grouping shows greater effectiveness. For the TPC-C workload which has high update skewness, SFS reduces the write cost by 77.4% under 90% utilization. Although uniform random workload without skewness is a worst case workload, SFS reduces the write cost by 27.9% under 90% utilization. This shows that SFS can effectively reduce the write cost for a variety of workloads.

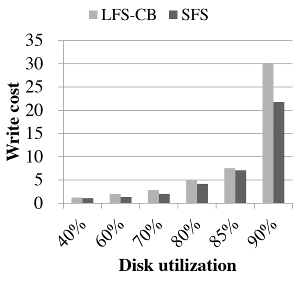
To see if the lower write costs in SFS will result in higher performance, throughput is also compared. As Figure 9 shows, SFS improves throughput of the TPC-C workload by 151.9% and that of uniform random workload by 18.5% under 90% utilization. It shows that the write cost reduction in SFS actually results in performance improvement.

4.3.2 Segment Utilization Distribution

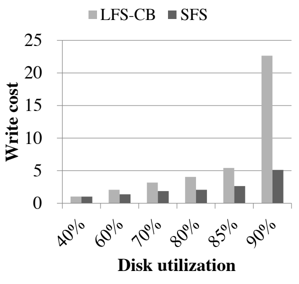
To further study why SFS significantly outperforms LFS-CB, we also compared the segment utilization distribution of SFS and LFS-CB. Segment utilization is calculated by dividing the number of live blocks in the segment by the number of total blocks per segment. After running a workload, the distribution is computed by measuring the utilizations of all non-dummy seg-



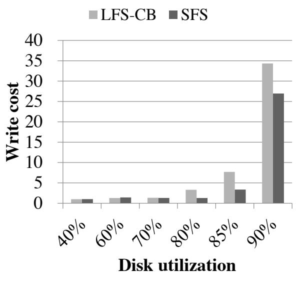
(a) Zipf



(b) Uniform random

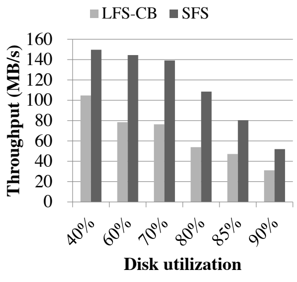


(c) TPC-C

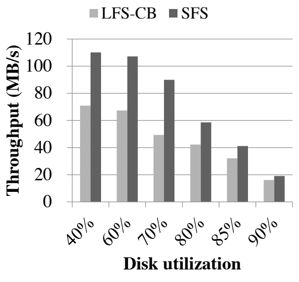


(d) RES

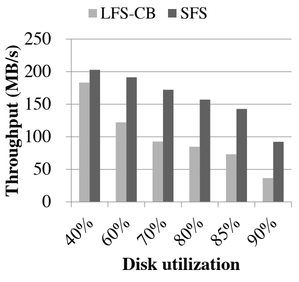
Figure 8: Write cost vs. disk utilization with SFS and LFS-CB on SSD-M.



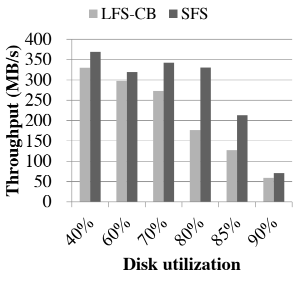
(a) Zipf



(b) Uniform random

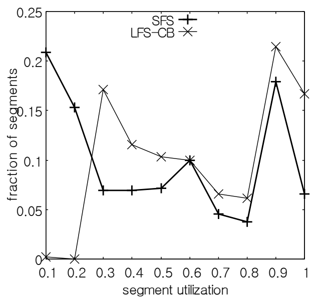


(c) TPC-C

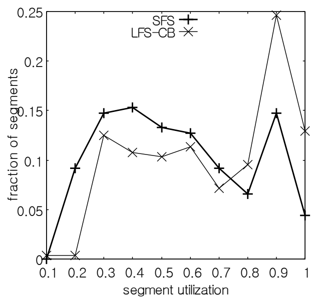


(d) RES

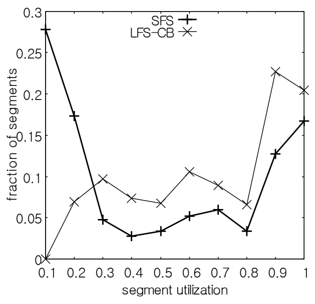
Figure 9: Throughput vs. disk utilization with SFS and LFS-CB on SSD-M.



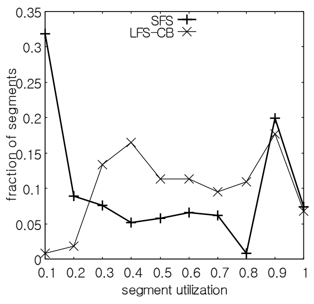
(a) Zipf



(b) Uniform random

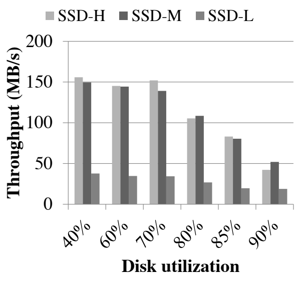


(c) TPC-C

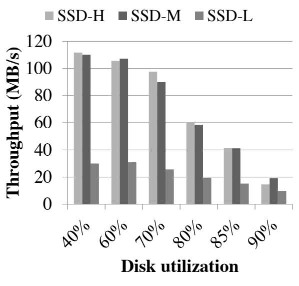


(d) RES

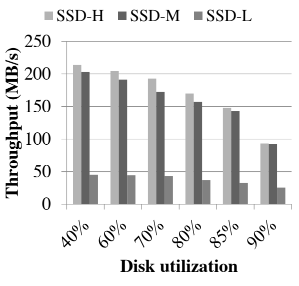
Figure 10: Segment utilization vs. fraction of segments. Disk utilization is 70%.



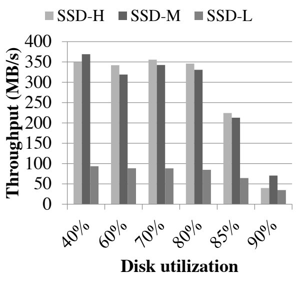
(a) Zipf



(b) Uniform random



(c) TPC-C



(d) RES

Figure 11: Throughput vs. disk utilization with SFS on different devices.

ments on the SSD. Since SFS continuously re-groups data blocks according to hotness, it is likely that a sharp bimodal distribution is formed. Figure 10 shows the segment utilization distribution when disk utilization is 70%. We can see the obvious bimodal segment distribution in SFS for all workloads except for the skewless uniform random workload. Even in the uniform random workload, the segment utilization of SFS is skewed to lower utilization. Under such bimodal distribution, the segment cleaner can select as victims those segments with few live blocks. For example, as shown in Figure 10a, SFS will select a victim segment with 10% utilization, while LFS-CB will select a victim segment with 30% utilization. In this case, the number of live blocks of a victim in SFS is just one-third of that in LFS-CB, thus the segment cleaner copies only one-third the amount of blocks. The reduced cleaning overhead results in a significant performance gap between SFS and LFS-CB. This experiment shows that SFS forms a sharp bimodal distribution of segment utilization by data block grouping, and reduces the write cost.

4.3.3 Effects of SSD Performance

In the previous sections, we showed that SFS can significantly reduce the write cost and drastically improve throughput on SSD-M. As shown in Section 2.2, SSDs have various performance characteristics. To see whether SFS can improve the performance on various SSDs, we compared throughput of the same workloads on SSD-H, SSD-M, and SSD-L in Figure 11. As shown in Table 1, SSD-H is ten-fold more expensive than SSD-L, the maximum sequential write performance of SSD-H is 4.5 times faster than SSD-L, and the 4 KB random write performance of SSD-H is more than 2,500 times faster than SSD-L. Despite the fact that these three SSDs show such large variances in performance and price, Figure 11 shows that SFS performs regardless of the random write performance. The main limiting factor is the maximum sequential write performance. This is because, except for updating superblock, SFS always generates large sequential writes to fully exploit the maximum bandwidth of SSD. The experiment shows that SFS can provide high performance even on mid-range or low-end SSD only if sequential write performance is high enough.

4.4 Comparison with Other File Systems

Up to now, we have analyzed how SFS performs under various environments with different workloads, disk utilization, and SSDs. In this section, we compared the performance of SFS using three other file systems, each with different block update policies: LFS-CB for *logging policy*, ext4 [25] for *in-place-update policy*, and btrfs [10] for *no-overwrite policy*. To enable btrfs' SSD

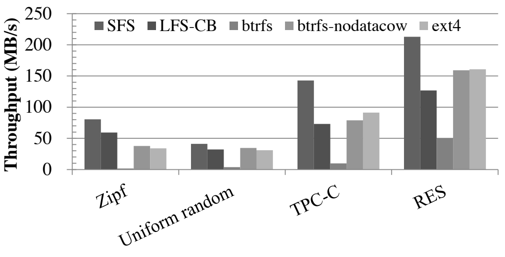
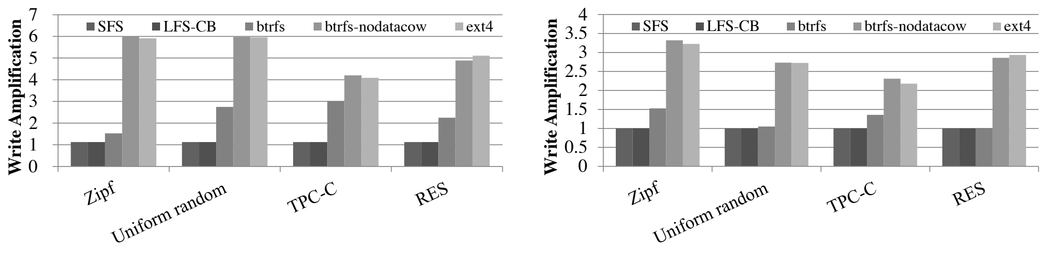


Figure 12: Throughput under different file systems.

optimization, btrfs was mounted in SSD mode. The in-place-update mode of btrfs is also tested with the `nodatacow` option enabled to further analyze the behavior of btrfs (hereafter btrfs-nodatacow). Four workloads were run on SSD-M with 85% disk utilization. To obtain the sustained performance, we measured 8 GB writing after 20 GB writing for aging.

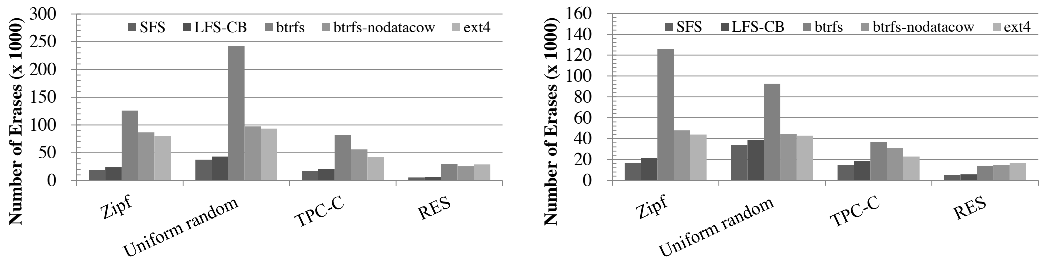
First, we compared throughput of the file systems in Figure 12. SFS significantly outperforms LFS-CB, ext4, btrfs, and btrfs-nodatacow for all four workloads. The average throughputs of SFS are higher than those of other file systems: 1.6 times for LFS-CB, 7.3 times for btrfs, 1.5 times for btrfs-nodatacow, and 1.5 times for ext4.

Next, we compared the write amplification that represents the garbage collection overhead inside SSD. We collected I/O traces issued by the file systems using `blktrace` [8] while running four workloads, and the traces were run on an FTL simulator, which we implemented, with two FTL schemes – (a) FAST [24] as a representative hybrid FTL scheme and (b) page-level FTL [17]. In both schemes, we configure a large block 32 GB NAND flash memory with 4 KB page, 512 KB block, and 10% over-provisioned capacity. Figure 13 shows write amplifications in FAST and page-level FTL for the four workloads processed by each file system. In all cases, write amplifications of log-structured file systems, SFS and LFS-CB, are very low: 1.1 in FAST and 1.0 in page-level FTL on average. This indicates that both FTL schemes generate 10% or less additional writings. Log-structured file systems collect and transform random writes at file level to sequential writes at LBA level. This results in optimal switch merge [24] in FAST and creates large chunks of contiguous invalid pages in page-level FTL. In contrast, in-place-update file systems, ext4 and btrfs-nodatacow, have the largest write amplification: 5.3 in FAST and 2.8 in page-level FTL on average. Since in-place-update file systems update a block in-place, random writes at file-level result in random writes at LBA-level. This contributes to high write amplification. Meanwhile, because btrfs never overwrites a block and allocates a new block for every update, it is likely to lower the average write amplification: 2.8 in FAST and



(a) FAST (b) Page Mapping

Figure 13: Write amplification with different FTL schemes.



(a) FAST (b) Page Mapping

Figure 14: Number of erases with different FTL schemes.

1.2 in page-level FTL on average.

Finally, we compared the number of block erases that determine the lifespan of SSD in Figure 14. As can be expected from the write amplification analysis, the number of block erases in SFS and LFS-CB are significantly lower than in all others. Since the segment cleaning overhead of SFS is lower than that of LFS-CB, the number of block erases in SFS is smallest: LFS-CB incurs totally 20% more block erases in FAST and page-level FTL. Erase counts of overwrite file systems, ext4 and btrfs-nodatacow, are significantly higher than that of SFS. In total, ext4 incurs 3.1 times more block erases in FAST and 1.8 times more block erases in page-level FTL. Similarly, total erase counts of btrfs-nodatacow are 3.4 times higher in FAST and 2.0 times higher in page-level FTL. Interestingly, btrfs incurs the largest number of block erases: in total, 6.1 times more block erases in FAST and 3.8 times more block erases in page-level FTL, and in worst case 7.5 times more block erases than SFS. Although the no-overwrite scheme in btrfs incurs lower write amplification compared to ext4 and btrfs-nodatacow, btrfs shows large overhead to support copy-on-write and manage fragmentation [21, 46] induced by random writes at file-level.

In summary, the erase count of the in-place-update file system is high because of high write amplification. That of the no-overwrite file system is also high due to the number of write requests from the file system, even at relatively low write amplification. The majority of the overhead comes from supporting no-overwrite and handling fragmentation in the file system. Fragmentation of the no-overwrite file system under random write is a widely known problem [21, 46]: successive random writes eventually move all blocks into arbitrary positions, and this makes all I/O access random at the LBA level. Defragmentation, which is similar to segment cleaning in a log-structured file system, is implemented [21, 1] to reduce the performance problem of fragmentation. Similarly to segment cleaning, it also has additional overhead to move blocks. In case of log-structured file systems, if we carefully choose segment size to be aligned with the clustered block size, write amplification can be minimal. In this case, the segment cleaning overhead is the major overhead that increases the erase count. SFS is shown to drastically reduce the segment cleaning overhead. It can also be seen that the write amplification and erase count of SFS are significantly lower than for all other file systems. Therefore, SFS can significantly increase the lifetime as well as the performance of SSDs.

5 Related Work

Flash memory based storage systems and log-structured techniques have received a lot of interests in both academia and industry. Here we only present the papers most related to our work.

FTL-level approaches: There are many FTL-level approaches to improve random write performance.

Among hybrid FTL schemes, FAST [24] and LAST [22] are representative. FAST [24] enhances random write performance by improving the log area utilization with flexible mapping in log area. LAST [22] further improves FAST [24] by separating random log blocks into hot and cold regions to reduce the full merge cost. Among page-level FTL schemes, DAC [13] and DFTL [14] are representative. DAC [13] clusters data blocks of the similar write frequencies into the same logical group to reduce the garbage collection cost. DFTL [14] reduces the required RAM size for the page-level mapping table by using dynamic caching. FTL-level approaches exhibit a serious limitation in that they depend almost exclusively on LBA to decide sequentiality, hotness, clustering, and caching. Such approaches deteriorate when a file system adopts a *no-overwrite* block allocation policy.

Disk-based log-structured file systems: There is much research to optimize log-structured file systems on conventional hard disks. In the *hole plugging* method [44], the valid blocks in victim segments are overwritten to the *holes*, i.e. invalid blocks, in other segments with a few invalid blocks. This reduces the copying cost of valid blocks in segment cleaning. However, this method is beneficial only under a storage media that allows in-place updates. Matthews et al. [26] proposed the *adaptive method* that combines cost-benefit policy and hole-plugging. It first estimates the cost of cost-benefit policy and hole-plugging respectively, and then adaptively selects the policy with the lower cost. However, their cost model is based on the performance characteristics of HDD, seek and rotational delay. WOLF [42] separates hot pages and cold pages into two different segment buffers according to the update frequency of data pages, and writes two segments to disk at once. This system works well only when hot pages and cold pages are roughly half and half, so that they can be separated into two segments. HyLog [43] uses a hybrid approach: logging for hot pages to achieve high write performance and overwrite for cold pages to reduce the segment cleaning cost. In HyLog, it is critical to estimate the ratio of hot pages to determine the update policy. However, similar to the adaptive method, its cost model is based on the performance characteristics of HDD.

Flash-based log-structured file systems: In embedded systems with limited CPU and main memory, specially designed file systems that directly access raw flash devices are commonly used. To handle the unique characteristics of flash memory including no in-place-update, wear-leveling and bad block management, these systems take the log-structured approach. JFFS2 [45], YAFFS2 [47], and UBIFS [41] are widely used flash-based log-structured file systems. In terms of segment cleaning, each uses a turn-based selection algorithm [45, 47, 41] that incorporates wear-leveling into the segment cleaning process. This consists of two phases, namely X and Y turns. In the X turn, it selects a victim segment using greedy policy without considering wear-leveling. During the Y turn, it probabilistically selects a full valid segment as a victim block for wear-leveling.

6 Conclusion and Future Work

In this paper, we proposed a next generation file system for SSD, SFS. It takes a log-structured approach which transforms the random writes at the file system into the sequential writes at the SSD, thus achieving high performance and also prolonging the lifespan of the SSD. Also, in order to exploit the skewness in I/O workloads, SFS captures the hotness semantics at file block level and utilizes these in grouping data eagerly on writing. In particular, we devised an iterative segment quantization algorithm for correct data grouping and also proposed the cost-hotness policy for victim segment selection. Our experimental evaluation confirms that SFS considerably outperforms existing file systems such as LFS, ext4, and btrfs, and prolongs the lifespan of SSDs by drastically reducing block erase count inside the SSD.

Another interesting question is the applicability of SFS for HDD. Though SFS was originally designed for targeting primarily for SSDs, its key techniques are agnostic to storage devices. While random write is more serious in SSD since it hurts the lifespan as well as performance, it hurts performance also in HDD due to increased seek-time. We did preliminary experiments to see if SFS is beneficial in HDD and got promising experimental results. As future work, we intend to explore the applicability of SFS for HDD in greater depth.

Acknowledgements

We thank the anonymous reviewers and our shepherd Keith Smith for their feedback and comments, which have substantially improved the content and presentation of this paper. This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education, Science and Technology (2011-0020520). This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MEST) (No. 2011-0027613).

References

[1] Linux 3.0. `http://kernelnewbies.org/Linux_3.0`.

[2] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *Proceeding of*

USENIX 2008 Annual Technical Conference, pages 57–70, Berkeley, CA, USA, 2008. USENIX Association.

[3] S. Akyürek and K. Salem. Adaptive block rearrangement. *ACM Trans. Comput. Syst.*, 13:89–121, May 1995.

[4] D. G. Andersen and S. Swanson. Rethinking Flash in the Data Center. *IEEE Micro*, 30:52–54, July 2010.

[5] L. Barroso. Warehouse-scale computing. In Keynote in the SIGMOD'10 conference, 2010.

[6] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis. BORG: block-reORGanization for self-optimizing storage systems. In *Proccedings of the 7th conference on File and storage technologies*, pages 183–196, Berkeley, CA, USA, 2009. USENIX Association.

[7] T. Blackwell, J. Harris, and M. Seltzer. Heuristic cleaning algorithms in log-structured file systems. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 23–23, Berkeley, CA, USA, 1995. USENIX Association.

[8] blktrace. `http://linux.die.net/man/8/blktrace`.

[9] L. Bouganim, B. n Jónsson, and P. Bonnet. uFLIP: Understanding Flash IO Patterns. In *Proceedings of the Conference on Innovative Data Systems Research*, CIDR '09, 2009.

[10] Btrfs. `http://btrfs.wiki.kernel.org`.

[11] S. D. Carson. A system for adaptive disk rearrangement. *Softw. Pract. Exper.*, 20:225–242, March 1990.

[12] F. Chen, D. A. Koufaty, and X. 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.

[13] M.-L. Chiang, P. C. H. Lee, and R.-C. Chang. Using data clustering to improve cleaning performance for plash memory. *Softw. Pract. Exper.*, 29:267–290, March 1999.

[14] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 229–240, New York, NY, USA, 2009. ACM.

[15] J. A. Hartigan and M. A. Wong. Algorithm AS 136: A K-Means Clustering Algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):pp. 100–108, 1979.

[16] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 19–19, Berkeley, CA, USA, 1994. USENIX Association.

[17] A. Kawaguchi, S. Nishioka, and H. Motoda. A flash-memory based file system. In *Proceedings of the USENIX 1995 Technical Conference*, TCON'95, pages 13–13, Berkeley, CA, USA, 1995. USENIX Association.

[18] J. Kim, J. M. Kim, S. Noh, S. L. Min, and Y. Cho. A space-efficient flash translation layer for CompactFlash systems. *IEEE Transactions on Consumer Electronics*, 48:366–375, May 2002.

[19] J. Kim, S. Seo, D. Jung, J. Kim, and J. Huh. Parameter-Aware I/O Management for Solid State Disks (SSDs). *To Appear in IEEE Transactions on Computers*, 2011.

[20] R. Konishi, K. Sato, and Y. Amagai. Filesystem support for Continuous Snapshotting. `http://www.nilfs.org/papers/ols2007-snapshot-bof.pdf`, 2007. Ottawa Linux Symposium 2007 BOFS material.

[21] J. Kára. Ext4, btrfs, and the others. In *Proceeding of Linux-Kongress and OpenSolaris Developer Conference*, pages 99–111, 2009.

[22] S. Lee, D. Shin, Y.-J. Kim, and J. Kim. LAST: locality-aware sector translation for NAND flash memory-based storage systems. *SIGOPS Oper. Syst. Rev.*, 42:36–42, October 2008.

[23] S.-W. Lee and B. Moon. Design of flash-based DBMS: an in-page logging approach. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 55–66, New York, NY, USA, 2007. ACM.

[24] 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 Trans. Embed. Comput. Syst.*, 6, July 2007.

[25] 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 of the Linux Symposium*, June 2007.

[26] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson. Improving the performance of log-structured file systems with adap-

tive methods. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, SOSP '97, pages 238–251, New York, NY, USA, 1997. ACM.

[27] S. Mitchel. *Inside the Windows 95 File System*. O'Reilly and Associates, 1997.

[28] NILFS2. `http://www.nilfs.org/`.

[29] R. Paul. Panelists ponder the kernel at Linux Collaboration Summit. `http://tinyurl.com/d7sht7`, 2009.

[30] QuestSoftware. Benchmark Factory for Databases. `http://www.quest.com/benchmark-factory/`.

[31] D. Roselli, J. R. Lorch, and T. E. Anderson. A comparison of file system workloads. In *Proceedings of USENIX Annual Technical Conference*, ATEC '00, pages 4–4, Berkeley, CA, USA, 2000. USENIX Association.

[32] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10:26–52, February 1992.

[33] C. Ruemmler and J. Wilkes. Disk Shuffling. Technical Report HPL-CSP-91-30, Hewlett-Packard Laboratories, October 1991.

[34] C. Ruemmler and J. Wilkes. UNIX disk access patterns. In *Proceedings of USENIX Winter 1993 Technical Conference*, page 405–420, 1993.

[35] M. Seltzer, K. Bostic, M. K. Mckusick, and C. Staelin. An implementation of a log-structured file system for UNIX. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, pages 3–3, Berkeley, CA, USA, 1993. USENIX Association.

[36] M. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File system logging versus clustering: a performance comparison. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 21–21, Berkeley, CA, USA, 1995. USENIX Association.

[37] E. Seppanen, M. T. O'Keefe, and D. J. Lilja. High performance solid state storage under Linux. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies*, MSST '10, pages 1–12, Washington, DC, USA, 2010. IEEE Computer Society.

[38] SNIA. Solid State Storage (SSS) Performance Test Specification (PTS) Enterprise Version 1.0. `http://www.snia.org/sites/default/files/SSS_PTS_Enterprise_v1.0.pdf`, 2011.

[39] R. Stoica, M. Athanassoulis, R. Johnson, and A. Ailamaki. Evaluating and repairing write performance on flash devices. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware*, DaMoN '09, pages 9–14, New York, NY, USA, 2009. ACM.

[40] Transaction Processing Performance Council. TPC Benchmark C. `http://www.tpc.org/tpcc/spec/tpcc_current.pdf`.

[41] UBIFS. Unsorted Block Image File System. `http://www.linux-mtd.infradead.org/doc/ubifs.html`.

[42] J. Wang and Y. Hu. A Novel Reordering Write Buffer to Improve Write Performance of Log-Structured File Systems. *IEEE Trans. Comput.*, 52:1559–1572, December 2003.

[43] W. Wang, Y. Zhao, and R. Bunt. HyLog: A High Performance Approach to Managing Disk Layout. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 145–158, Berkeley, CA, USA, 2004. USENIX Association.

[44] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *ACM Trans. Comput. Syst.*, 14:108–136, February 1996.

[45] D. Woodhouse. JFFS : The Journalling Flash File System. In *Proceedings of the Ottowa Linux Symposium*, 2001.

[46] M. Xie and L. Zefan. Performance Improvement of Btrfs. In *LinuxCon Japan*, 2011.

[47] YAFFS. Yet Another Flash File System. `http://www.yaffs.net/`.

[48] ZFS. `http://opensolaris.org/os/community/zfs/`.

FIOS: A Fair, Efficient Flash I/O Scheduler*

Stan Park and Kai Shen
Department of Computer Science, University of Rochester
{park, kshen}@cs.rochester.edu

Abstract

Flash-based solid-state drives (SSDs) have the potential to eliminate the I/O bottlenecks in data-intensive applications. However, the large performance discrepancy between Flash reads and writes introduces challenges for fair resource usage. Further, existing fair queueing and quanta-based I/O schedulers poorly manage the I/O anticipation for Flash I/O fairness and efficiency. Some also suppress the I/O parallelism which causes substantial performance degradation on Flash. This paper develops FIOS, a new Flash I/O scheduler that attains fairness and high efficiency at the same time. FIOS employs a fair I/O timeslice management with mechanisms for read preference, parallelism, and fairness-oriented I/O anticipation. Evaluation demonstrates that FIOS achieves substantially better fairness and efficiency compared to the Linux CFQ scheduler, the SFQ(D) fair queueing scheduler, and the Argon quanta-based scheduler on several Flash-based storage devices (including a CompactFlash card in a low-power wimpy node). In particular, FIOS reduces the worst-case slowdown by a factor of 2.3 or more when the read-only SPECweb workload runs together with the write-intensive TPC-C.

1 Introduction

NAND Flash devices [1, 20, 24] are widely used as solid-state storage on conventional machines and low-power wimpy nodes [2, 6]. Compared to mechanical disks, they deliver much higher I/O performance which can alleviate the I/O bottlenecks in critical data-intensive applications. Emerging non-volatile memory (NVRAM) technologies such as phase-change memory [10, 12], memristor, and STT-MRAM promise even better performance. However, these NVMs under today's manufacturing technologies still suffer from low space density (or high \$/GB) and stability/durability problems. Until these issues are resolved sometime in the future, NAND Flash devices will likely remain the dominant solid-state storage in computer systems.

While Flash-based storage devices may offer substantially improved I/O performance over mechanical disks, there are critical limitations with respect to writes. First, Flash suffers from an erase-before-write limitation. That is, in order to overwrite a previously written location, the said location must first be erased before writing the new data. Further aggravating the problem is that the erasure granularity is typically much larger (64–256$\times$) than the basic I/O granularity (2–8 KB). This leads to a large read/write speed discrepancy—Flash reads can be one or two orders of magnitude faster than writes. This is very different from mechanical disks on which read/write performance are both dominated by seek/rotation delays and exhibit similar characteristics.

For a concurrent workload with a mixture of readers and synchronous writers running on Flash, readers may be blocked by writes with substantial slowdown. This means unfair resource utilization between readers and writers. In extreme cases, it may present vulnerability to denial-of-service attacks—a malicious user may invoke a workload with a continuous stream of writes to block readers. At the opposite end, strictly prioritizing reads over writes might lead to unfair (and sometimes extreme) slowdown for applications performing synchronous writes. Synchronous writes are essential for applications that demand high data consistency and durability, including databases, data-intensive network services [28], persistent key-value store [2], and periodic state checkpointing [19].

With important implications on performance and reliability, Flash I/O fairness warrants first-class attention in operating system I/O scheduling. Conventional scheduling methods to achieve fairness (like fair queueing [5, 18] and quanta-based scheduling [3, 36]) fail to recognize unique Flash characteristics like substantial read-blocked-by-write. In addition, I/O anticipation (temporarily idling the device in anticipation of a soon-arriving desirable request) is sometimes necessary to maintain fair resource utilization. While I/O anticipation was proposed as a performance-enhancing seek-reduction technique for mechanical disks [17], its role for maintaining fairness has been largely ignored. Finally, quanta-based scheduling schemes [3, 36] typically suppress the I/O parallelism between concurrent tasks,

*This work was supported in part by the National Science Foundation (NSF) grant CCF-0937571, NSF CAREER Award CCF-0448413, a Google Research Award, and an IBM Faculty Award.

which substantially degrades the I/O efficiency on Flash devices with internal parallelism.

This paper presents a new operating system I/O scheduler (called *FIOS*) that achieves fair Flash I/O while attaining high efficiency at the same time. Our scheduler uses timeslice management to achieve fair resource utilization under high I/O load. We employ read preference to minimize read-blocked-by-write in concurrent workloads. We exploit device-level parallelism by issuing multiple I/O requests simultaneously when fairness is not violated. Finally, we manage I/O anticipation judiciously such that we achieve fairness with limited cost of device idling.

We implemented our scheduler in Linux and demonstrated our results on multiple Flash devices including three solid-state disks and a CompactFlash card in a low-power wimpy node. Our evaluation employs several application workloads including the SPECweb workload on an Apache web server, TPC-C workload on a MySQL database, and the FAWN Data Store developed specifically for low-power wimpy nodes [2]. Our empirical work also uncovered a flaw in the current Linux's inconsistent management of synchronous writes across file system and I/O scheduler layers.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 characterizes key challenges for supporting Flash I/O fairness and efficiency that motivate our work. Section 4 presents the design of our FIOS scheduler for Flash storage devices. Section 5 describes some implementation notes and Section 6 illustrates our experimental evaluation. Section 7 concludes this paper with a summary of our findings.

2 Related Work

There are significant recent research interests in I/O performance characterization of Flash-based storage devices. Agrawal *et al.* [1] discussed the impact of block erasure (before writes) and parallelism to the performance of Flash-based SSDs. Polte *et al.* [31] found that Flash reads are substantially faster than writes. Past studies identified abnormal performance issues due to read/write interference and storage fragmentation [7], as well as erasure-induced variance of Flash write latency [9]. There is also a recognition on the importance of internal parallelism to the Flash I/O efficiency [8, 30] while our past work identified that the effects of parallelism depend on specific firmware implementations [30]. Previous Flash I/O characterization results provide motivation and foundation for Flash I/O scheduling work in this paper.

Recent research has investigated operating system techniques to manage Flash-based storage. File system work [11, 23, 25] has attempted to improve the sequential write patterns through the use of log-structured file systems. These efforts are orthogonal to our research on Flash I/O scheduling. New I/O scheduling heuristics were proposed to improve Flash I/O performance. In particular, write bundling [21], write block preferential [14], and page-aligned request merging/splitting [22] help match I/O requests with the underlying Flash device data layout. The effectiveness of these write alignment techniques, however, is limited on modern SSDs with write-order-based block mapping. Further, previous Flash I/O schedulers have paid little attention to the issue of fairness.

Conventional I/O schedulers are largely designed to mitigate the high seek and rotational costs in mechanical disks, through elevator-style I/O request ordering and anticipatory I/O [17]. Quality-of-service objectives (like meeting task deadlines) were also considered in I/O scheduling techniques, including Facade [27], Reddy *et al.* [33], pClock [16], and Fahrrad [32]. Fairness was not a primary concern in these techniques and they cannot address the fairness problems in Flash storage devices.

Fairness-oriented resource scheduling has been extensively studied in the literature. The original fair queueing approaches including Weighted Fair Queueing (WFQ) [13], Packet-by-Packet Generalized Processor Sharing (PGPS) [29], and Start-time Fair Queueing (SFQ) [15] take virtual time-controlled request ordering over several task queues to maintain fairness. While they are designed for network packet scheduling, later fair queueing approaches like YFQ [5] and SFQ(D) [18] are adapted to support I/O resources. In particular, they allow the flexibility to re-order and parallelize I/O requests for better efficiency. Alternatively, I/O fair queueing can be achieved using dedicated per-task quanta (as in Linux CFQ [3] and Argon [36]) and credits (as in the SARC rate controller [37]). Achieving fairness and efficiency on Flash storage, however, must address unique Flash I/O characteristics like read/write performance asymmetry and internal parallelism. A proper management of I/O anticipation for fairness is also necessary.

3 Challenges and Motivation

We characterize key challenges for supporting Flash I/O fairness and maintaining high efficiency at the same time. They include effects of inherent device characteristics (read/write asymmetry and internal parallelism) as well as behavior of operating system I/O schedulers (role of I/O anticipation). These results and analysis serve as both background and motivation for our new I/O scheduling design.

Experiments in this section and the rest of the paper will utilize the following Flash-based storage devices—

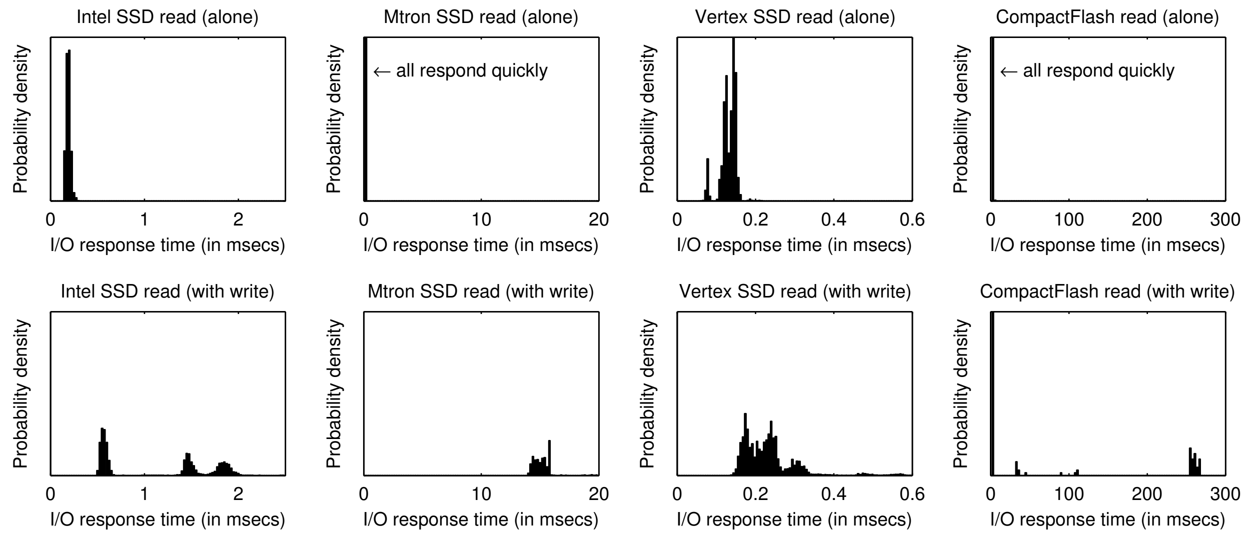


Figure 1: Distribution of 4 KB read response time on four Flash-based storage devices. The first row shows the read response time when a read runs alone. The second row shows the read performance at the presence of a concurrent 4 KB write. The two figures in each column (for one drive) use the same X-Y scale and they can be directly compared. Figures across different columns (for different drives) necessarily use different X-Y scales due to differing drive characteristics. We intentionally do not show the quantitative Y values (probability densities) in the figures because these values have no inherent meaning and they simply depend on the width of each bin in the distribution histogram.

- An Intel X25-M Flash-based SSD released in 2009. This drive uses multi-level cells (MLC) in which a particular cell is capable of storing multiple bits of information.
- An Mtron Pro 7500 Flash-based SSD, released in 2008, using single-level cells (SLC).
- An OCZ Vertex 3 Flash-based SSD, released in 2011, using MLC. This drive employs the SandForce controller which supports new write acceleration techniques such as online compression.
- A SanDisk CompactFlash drive on a 6-Watts "wimpy" node similar to those employed in the FAWN array [2].

Read/Write Fairness Our first challenge to Flash I/O fairness is that Flash writes are often substantially slower than reads and a reader may experience excessive slowdown at the presence of current writes. We try to understand this by measuring the read/write characteristics of the four Flash devices described above. To acquire the native device properties, we bypass the memory buffer, operating system I/O scheduler, and the device write cache in the measurements. We also use incompressible data in the I/O measurement to assess the baseline performance for the Vertex drive (whose SandForce controller performs online compression).

Our measurement employs 4 KB reads or writes to random storage locations. Figure 1 illustrates the read response time distribution in two cases—read alone and read at the presence of a concurrent write. Comparing that with the read-alone performance (first row), we find that a Flash read can experience one or two orders of magnitude slowdown while being blocked by a concurrent write. Further, the Flash read response time becomes much less stable (or more unpredictable) when blocked by a concurrent write. One exception to this finding is the Vertex drive with the SandForce controller. Writes on this drive is only modestly slower than reads and therefore the read-block-by-write effect is much less pronounced on this drive than on others.

We further examine the fairness between two tasks—a *reader* that continuously performs 4 KB reads to random locations (issues another one immediately after the previous one completes) and a *writer* that continuously performs synchronous 4 KB writes to random locations. Figure 2 shows the slowdown ratios for reads and writes during a concurrent execution. Results show that the write slowdown ratios are close to one on all Flash storage devices, indicating that the write performance in the concurrent execution is similar to the write-alone performance. However, reads experience 7×, 157×, 2×, and 42× slowdown on the Intel SSD, Mtron SSD, Vertex SSD, and the low-power CompactFlash respectively.

Existing fairness-oriented I/O schedulers [3, 5, 18, 36, 37] do not recognize the Flash read/write performance asymmetry. Consequently they provided no support to

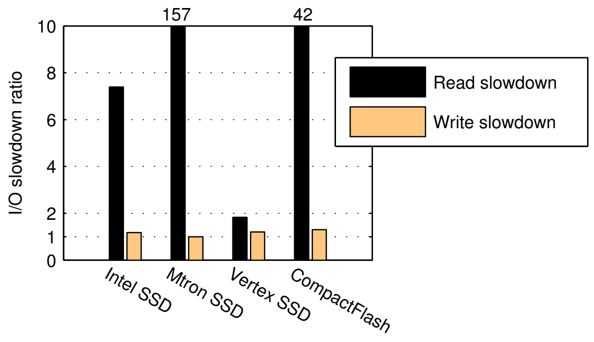


Figure 2: Slowdown of random 4 KB reads and writes in a concurrent execution. The *I/O slowdown ratio* for read (or write) is the I/O latency normalized to that when running alone.

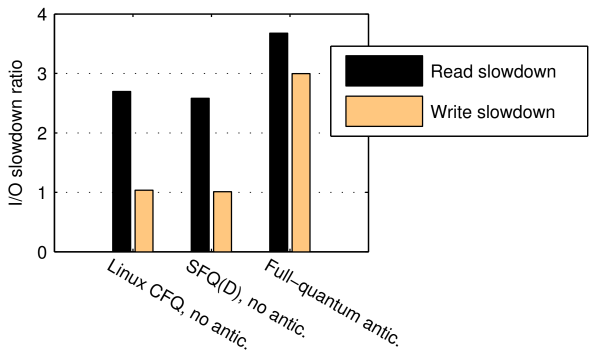


Figure 3: Fairness of different I/O anticipation approaches for concurrent reader/writer on the Intel SSD.

address the problem of excessive read-blocked-by-write on Flash.

Role of I/O Anticipation I/O anticipation (temporarily idling the device in anticipation of a soon-arriving desirable request) was proposed as a performance-enhancing seek-reduction technique for mechanical disks [17]. However, its performance effects on Flash are largely negative because the cost of device idling far outweighs limited benefit of I/O spatial proximity. Due to the lack of performance gain on Flash, the Linux CFQ scheduler disables I/O anticipation for non-rotating storage devices like Flash. Fair queueing approaches like YFQ [5] and SFQ(D) [18] also provide no support for I/O anticipation.

However, I/O anticipation is sometimes necessary to maintain fair resource utilization. Without anticipation, unfairness may arise due to the prematurely switching task queues before the allotted I/O quantum is fully utilized (in quanta-based scheduling) or the premature advance of virtual time for "inactive tasks" (in fair queueing schedulers). Consider the simple example of a concurrent run involving a reader and a writer. After servicing a read, the only queued request at the moment is a write and therefore a work-conserving I/O scheduler will issue it. This breaks up the allotted quantum for the reader. Even if the reader issues another read after a short thinktime, it would be blocked by the outstanding write.

At the opposite end, the quanta-based scheduling in Argon [36] employs aggressive I/O anticipation such that it is willing to wait through a task queue's full quantum even if few requests are issued. Such excessive I/O anticipation can lead to long idle time and drastically reduce performance on Flash storage if useful work could otherwise have been accomplished. Particularly for fast Flash storage, a few milliseconds are often sufficient for completing a significant amount of work.

We run a simple experiment to demonstrate the fairness and efficiency effects of improper I/O anticipation on Flash. We run a reader and a writer concurrently on the Intel SSD. Each task induces some thinktime between I/O such that the thinktime time is approximately equal to its I/O device usage time. Figure 3 shows the reader/writer slowdown under three I/O scheduling approaches. Implementation details of the schedulers are provided later in Section 5. The Linux CFQ and SFQ(D) do not support I/O anticipation which leads to poor fairness between the reader and writer. The full-quantum anticipation exhibits better fairness (similar reader/writer slowdown) but this is achieved at excessive slowdown for both reader and writer. Such fairness is not worthwhile.

While our discussion above uses the example of a reader running concurrently with a writer, the fairness implication of I/O anticipation generally applies to concurrent tasks with requests of differing resource usage. For instance, similar fairness problems with no I/O anticipation or over-aggressive anticipation can arise when a task making 4 KB reads runs concurrently with a task making 128 KB reads.

Parallelism vs. Fairness Flash-based SSDs have some built-in parallelism through the use of multiple channels. Within each channel, each Flash package may have multiple planes which are also parallel. Figure 4 shows the efficiency of Flash I/O parallelism for 4 KB reads and writes on our Intel, Mtron, and Vertex SSDs. We observe that the parallel issuance of multiple reads to an SSD may lead to throughput enhancement. The speedup is modest (about 30%) for the Mtron SLC drive but substantial (up to 7-fold and 4-fold) for the Intel and Vertex MLC drives. On the other hand, writes do not seem to benefit from I/O parallelism on the Intel and Mtron drives while write parallelism on the Vertex drive can have up to 3-fold speedup. We also experimented with parallel I/O at larger (>4 KB) sizes and we found that the speedup of

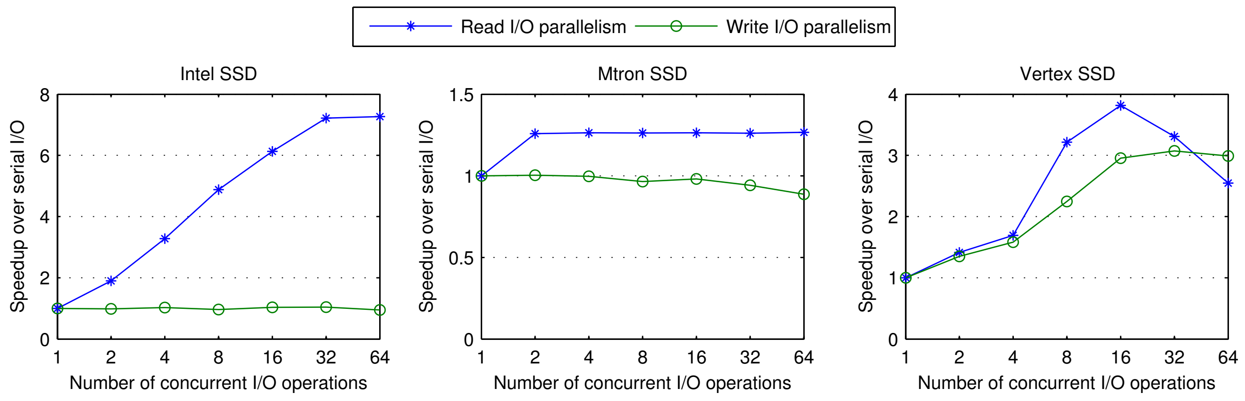


Figure 4: Efficiency of I/O parallelism for 4 KB reads and writes on three Flash-based SSDs.

parallel request issuance is less substantial for large I/O requests. A possible explanation is that a single large I/O request may already benefit from the internal device parallelism and therefore parallel request issuance will see less additional efficiency gain.

The internal parallelism on Flash-based SSDs has significant implication on fairness-oriented I/O scheduling. In particular, the quanta-based schedulers (like Linux CFQ [3] and Argon [36]) only issue I/O requests from one task queue at a time, which limits parallelism. The rationale is probably to ease the accounting and allocation of device time usage for each queue. However, the suppression of I/O parallelism in these schedulers may lead to substantial performance degradation on Flash. A desired Flash I/O scheduler must exploit device-level parallelism by issuing multiple I/O requests simultaneously while ensuring fairness at the same time.

4 FIOS Design

In a multiprocessing system, many resource principals simultaneously compete for the shared I/O resource. The scheduler should regulate I/O in such a way that accesses are fair. When the storage device time is the bottleneck resource in the system, fairness is the case that each resource principal acquires an equal amount of device time. When the storage device is partially loaded, the critical problem is that a read blocked by a write experiences far worse slowdown than a write blocked by a read. Such worst-case slowdown should be minimized.

Practical systems may desire fairness for different kinds of resource principals. For example, a general-purpose operating system may desire fairness support among concurrent processes. A server system may need fairness across simultaneously running requests [4, 34]. A shared hosting platform may want fairness across multiple virtual machines [26]. Our design of fair Flash I/O scheduling and much of our implementation can be generally applied to supporting arbitrary resource principals. When describing the FIOS design, we use the term *task* to represent the resource principal that receives the fairness support in a concurrent execution.

Our I/O scheduler, FIOS, tries to achieve fairness while attaining high efficiency at the same time. Based on our evaluation and analysis in Section 3, our scheduler contains four techniques. We first provide a fair timeslice management that allows timeslice fragmentation and concurrent request issuance (Section 4.1). We then support read preference to minimize the read-blocked-by-write situations (Section 4.2). We further enable concurrent issuance of requests to maximize the efficiency of device-level parallelism (Section 4.3). Finally, we devise limited I/O anticipation to maintain fairness at minimal device idling cost (Section 4.4).

4.1 Fair Timeslice Management

FIOS builds around a fairness mechanism of equal timeslices which govern the amount of time a task has access to the storage device. As each task is given equal time-based access to the storage device, the disparity between read and write access latency of Flash cannot lead to unequal device usage between tasks. In addition, using timeslices provides an upper bound on how long a task may have access to the storage device, ensuring that no task will be starved indefinitely.

Our I/O timeslices are reminiscent of the I/O quanta in quanta-based fairness schedulers like Linux CFQ [3] and Argon [36]. However, the previous quanta-based schedulers suffer two important limitations that make them unsuitable for Flash fairness and efficiency.

- First, their I/O quanta do not allow fragmentation—a task must use its current quantum continuously or it will have to wait for its next quantum in the round-robin order. The rationale (on mechanical disk storage devices) was that long continuous run by a

single task tends to require less disk seek and rotation [36]. But for a task that performs I/O with substantial inter-I/O thinktime, this design leaves two undesirable choices—either its quantum ends prematurely so the remaining allotted resource is forfeited (as in Linux CFQ) or the device idles through a task's full quantum even if few requests are issued (as in Argon).
- Second, the previous quanta-based schedulers only allow I/O requests from one task to be serviced at a time. This was a reasonable design decision for individual mechanical disks that do not possess internal parallelism. It also has the advantage of easy resource accounting for each task. However, this mechanism suppresses Flash I/O parallelism and consequently hurts I/O efficiency.

To address these problems, FIOS allows I/O timeslice fragmentation and concurrent request issuance from multiple tasks. Specifically, we manage timeslices in an epoch-based fashion. An epoch is defined by a collection of equal timeslices, one per task; the I/O scheduler should achieve fairness in each epoch. After an I/O completion, the task's remaining timeslice is decremented by an appropriate I/O cost. The cost is the elapsed time from the I/O issuance to its completion when the storage device is dedicated to this request in this duration. The cost accounting is more complicated in the presence of parallel I/O from multiple tasks, which will be elaborated in Section 4.3. A currently active task does not forfeit its remaining timeslice should another task be selected for service by the scheduler. In other words, the timeslice of a task can be consumed over several non-contiguous periods within an epoch. Once a task has consumed its entire timeslice, it must wait until the next epoch at which point its timeslice is refreshed.

The current epoch ends and a new epoch begins when either 1) there is no task with non-zero remaining timeslice in the current epoch; or 2) all tasks with non-zero remaining timeslices make no I/O request. Fairness must be maintained in the case of deceptive idleness [17]. Specifically, the I/O scheduler may observe a short idle period from a task between two consecutive I/O requests it makes. A fair-timeslice epoch should not end at such deceptive idleness if the task has non-zero remaining timeslice. This is addressed through fairness-oriented I/O anticipation elaborated in Section 4.4.

4.2 Read/Write Interference Management

Our preliminary evaluation in Section 3 shows strong interference between concurrent reads and writes on some of the Flash drives, an effect also observed by others [7]. Considering that reads are faster than writes, reads suffer more dramatically from such interference while the impact on writes appears marginal. A concurrent write not only slows down reads, it also disrupts device-level read parallelism which leads to further efficiency loss. Part of our fairness goal is to minimize the worst-case task slowdown. For such fairness, we adopt a policy of read preference combined with write blocking to reduce the read-interfered-by-write occurrences. Such a policy gives preference to shorter jobs, which tends to produce faster mean response time than a scheduler that is indiscriminate of job service time. This is a side benefit beyond minimizing the worst-case slowdown.

When both read and write requests are queued in the I/O scheduler, our policy of read preference will allow read requests to be issued first. To further avoid interference from later-issued writes, we block all write requests until outstanding reads are completed. Under this approach, a read is only blocked by a write when the read arrives at the I/O scheduler after the write has already been issued. This is due to the non-preemptibility of I/O. Both read preference and write blocking lead to additional queuing time for writes. Fortunately, because reads are serviced quickly, the additional queueing time the write request experiences is typically small compared to the write service time. Note that the read preference mechanism is still governed by the epoch-based timeslice enforcement, which serves as an ultimate preventer of write starvation.

Our preliminary evaluation in Section 3 also shows that while the read/write interference is very strong on some drives, it is quite modest on the Vertex SSD. On such a drive, the benefit of read preference and write blocking is modest and it may be outweighed by its drawbacks of possible write starvation and suppressing the mixed read/write parallelism. Therefore the read/write interference management is an optional feature in FIOS that can be disabled for drives that do not exhibit strong read/write interference.

4.3 I/O Parallelism

Many Flash-based solid-state drives contain internal parallelism that allows multiple I/O requests to be serviced at the same time. To achieve high efficiency and exploit the parallel architecture in Flash, multiple I/O requests should be issued to the Flash device in parallel when fairness is not violated. After issuing an I/O request to the storage device, FIOS searches for additional requests which may be queued, possibly from other tasks. Any I/O requests that are found are issued as long as the owner tasks have enough remaining timeslices and the read/write interference management (if enabled) is observed.

I/O parallelism allows multiple tasks to access the storage device concurrently, which complicates the accounting of I/O cost. In particular, a task should not be billed by the full elapsed time from its request issuance to completion if requests from other tasks are simultaneously outstanding on the storage device. The ideal cost accounting for an I/O request should exclude the request queueing time at the device during which it waits for other requests and it does not consume the bottleneck resource. A precise accounting, however, is difficult without the device-level knowledge of resource sharing between multiple outstanding requests.

We support two approaches for I/O cost accounting under parallelism. In the first approach, we calibrate the elapsed time of standalone read/write requests at different data sizes and use the calibration results to assign the cost of an I/O request online depending on its type (read or write) and size. Our implementation further assumes a linear model (typically with a substantial nonzero offset) between the cost and data size of an I/O request. Therefore we only need to calibrate four cases (read 4 KB, read 128 KB, write 4 KB, and write 128 KB) and use the linear model to estimate read/write costs at other data sizes. In practice, such calibration is performed once for each device, possibly at the device installation time. Note that the need of request cost estimation is not unique to our scheduler. Start-time Fair Queueing schedulers [15, 18] also require a cost estimation for each request when it just arrives (for setting its start and finish tags).

When the calibrated I/O costs are not available, our scheduler employs a backup approach for I/O cost accounting. Here we make the following assumption about the sharing of cost for parallel I/O. During a time period when the set of outstanding I/O requests on the storage device remains unchanged (no issuance of a new request or completion of an outstanding request), all outstanding I/O requests equally share the device usage cost in this time period. This is probabilistically true when the internal device scheduling and operation is independent of the task owning the request. Such an assumption allows us to account for the cost of parallel I/O with only information available to the operating system. Since the device parallelism may change during a request's execution, an accurate accounting of a request's execution parallelism would require carefully tracking the device parallelism throughout its execution duration. For simplicity, we use the device parallelism at the time of request issuance to represent the request execution parallelism. Specifically, the I/O cost is calculated as

$$\text{Cost} = \frac{T_{\text{elapsed}}}{P_{\text{issuance}}} \tag{1}$$

where T_{elapsed} is the request's elapsed time from its issuance to its completion, and P_{issuance} is the number of outstanding requests (including the new request) at the issuance time.

4.4 I/O Anticipation for Fairness

Between two consecutive I/O requests made by a task, the I/O scheduler may observe a short idle period. This idle period is unavoidable because it takes non-zero time for the task to wake up and issue another request. Such an idleness is deceptive for tasks that continuously make synchronous I/O requests. The deceptive idleness can be addressed by I/O anticipation [17], which idles the storage device in anticipation of a soon-arriving new I/O request. On mechanical disks, I/O anticipation can substantially improve the I/O efficiency by reducing the seek and rotation overhead. In contrast, I/O spatial proximity has much less benefit for Flash storage. Therefore I/O anticipation has a negative performance effect and it must be used judiciously for the purpose of maintaining fairness. Below we describe two important decisions about fairness-oriented I/O anticipation on Flash—When to anticipate? How long to anticipate?

When to anticipate? Anticipation is always considered when a request is just completed. We call the task that owns the just completed request the *anticipating task*.

Deceptive idleness may break fair timeslice management when it prematurely triggers an epoch switch while the anticipating task will quickly process the just completed I/O request and issue another one soon. I/O anticipation should be utilized to remedy such a fairness violation. Specifically, while an epoch would normally end if there is no outstanding I/O request from a task with non-zero remaining timeslice, we initiate an anticipation before the epoch switch if the anticipating task has non-zero remaining timeslice. In this case the anticipation target can be either a read or write, though it is more commonly write since writers are more likely delayed to the end of an epoch under read preference.

Deceptive idleness may also break read preference. When there are few tasks issuing reads, there may be instances when no read request is queued. In order to facilitate read preference, I/O anticipation is necessary after completing a read request. If a read request has just been completed, we anticipate for another read request to arrive shortly. We do so rather than immediately issuing a write to the device that may block later reads.

How long to anticipate? I/O anticipation duration must be bounded in case the anticipated I/O request never arrives. For maximum applicability and robustness, the system should not assume any application hints or predictor of the application inter-I/O thinktime. For seek-reduction on mechanical disks, the I/O anticipation bound is set to roughly the time of a disk I/O operation which leads to competitive performance compared to the optimal offline I/O anticipation. In practice, this is often set to 6 or 8 milliseconds. Our I/O anticipation bound must be different for two reasons. First, the original anticipation bound addresses the device idling's tradeoff with performance gain of seek reduction. Anticipation has a negative performance effect on Flash and we instead target the different tradeoff with maintaining fairness. Second, the Flash I/O service time is much smaller than that of a disk I/O operation. This exacerbates the cost of anticipation-induced device idling on Flash.

FIOS sets the I/O anticipation bound according to a configurable threshold of tolerable performance loss for maintaining fairness. This threshold, α, indicates the maximum proportion of time FIOS idles the device (while there is pending work) to anticipate for fairness. Specifically, when the deceptive idleness is about to break fairness, we anticipate for an idling time bound of $T_{\text{service}} \cdot \frac{\alpha}{1-\alpha}$, where T_{service} is the average service time of an I/O request for the anticipating task. This ensures that the maximum device idle time is no more than α proportion of the total device time in a sequence of

$$\text{I/O} \rightarrow \text{anticipation} \rightarrow \text{I/O} \rightarrow \text{anticipation} \rightarrow \cdots$$

In our implementation, FIOS maintains the per-request I/O service time T_{service} for each task using an exponentially-weighted moving average of past request statistics. FIOS sets $\alpha = 0.5$ by default.

Anticipation-induced device idling consumes device time and its cost must be properly accounted and attributed. We charge the anticipation cost to the timeslice of the anticipating task.

5 Implementation Notes

We implemented our FIOS scheduler with the techniques of fair timeslice management, read preference, I/O parallelism, and I/O anticipation for fairness on Linux 2.6.33.4. As part of a general-purpose operating system, our prototype provides fairness to concurrent processes. This implementation can be easily extended to support request-level fairness in a server system [4,34] or virtual machine fairness in a shared hosting platform [26].

Our I/O anticipation may sometimes desire a very short timer (a few hundred microseconds). The default Linux I/O schedulers use the kernel tick-based timer. Specifically with 1000 Hz kernel ticks, the minimum timer is 1 millisecond. Further, because the kernel ticks are not synchronized with the timer setup, the next tick may occur right after the timer is set. This means that setting the timer to fire at the next tick may sometimes lead to almost no anticipation. Our recent research [35] showed that this already happened to some production versions of Linux with coarse-grained tick timers. Our FIOS implementation instead uses the Linux high-resolution timer that can be supported by the processor hardware counter overflow interrupts. This allows us to set precise, fine-grained anticipation timers.

For comparison purposes, we implemented two alternative fairness-oriented I/O schedulers in our experimental platform. The first alternative is SFQ(D) [18], which is based on the Start-time Fair Queueing approach [15] but also allows concurrent request issuance for I/O efficiency. The concurrency is controlled by a depth parameter *D*. We set the depth to 32 which allows sufficient I/O parallelism in all our experiments. The SFQ(D) scheduler requires a cost estimation for each request when it just arrives (for setting its start and finish tags in SFQ(D)). In our implementation, we estimate a read's cost as the average read service time on the device; similarly, we estimate the cost of a write as the average write service time on the device.

The second alternative is a quanta-based I/O scheduler like the one employed in Argon [36]. This approach puts a high priority on achieving fair resource use (even if some tasks only have partial I/O load). All tasks take round robin turns of I/O quanta. Each task has exclusive access to the storage device within its quantum. Once an I/O quantum begins, it will last to its end, regardless of how few requests are issued by the corresponding task. However, a quantum will not begin, if no request from the corresponding task is pending.

The Linux CFQ, our FIOS scheduler, and the quanta scheduler all use the concept of per-task timeslice or quantum. In the Linux CFQ, the default timeslice is 100 milliseconds, with minor adjustment according to task priorities. Our FIOS and quanta scheduler implementations follow the same setting of per-task timeslice/quantum.

During our empirical work, we discovered a flaw in Linux that it inconsistently manages synchronous writes across the file system and I/O scheduler layers. Specifically, a synchronous operation at the file system level (such as a write on an `O_SYNC`-opened file and I/O as part of a `fsync()` call) is not necessarily considered to be synchronous at the I/O scheduler. Note that this inconsistency does not lead to wrong synchronous I/O semantics to the application since the file system will force a wait on the I/O completion before returning to the application. However, being treated as asynchronous I/O at the I/O scheduler means that they are scheduled with lowest priority, leading to excessive delay by the applications who perform synchronous I/O. We fixed this problem by patching `mpage_writepage()` functions in the Linux kernel so that file system-level synchronous operations are properly considered synchronous I/O at the scheduler.

We perform experiments on the ext4 file system. The ext4 file system uses very fine-grained file timestamps (in nanoseconds) so that each file write always leads to a new modification time and thus triggers an additional metadata write. This is unnecessarily burdensome to many write-intensive applications. We revert back to file timestamps in the granularity of seconds (which is the default in Linux file systems that do not make customized settings). In this case, at most one timestamp metadata write per second is needed regardless how often the file is modified.

We also found that the file system journaling writes made the evaluation results less stable and harder to interpret. Therefore we disabled the journaling in our experiments. We do not believe this setup choice affects the fundamental results of our evaluation.

6 Experimental Evaluation

We compare FIOS's fairness and efficiency against three alternative fairness-oriented I/O schedulers—Linux CFQ scheduler [3], SFQ(D) start-time fair queueing with a concurrency depth [18], and a quanta-based I/O scheduler similar to the one employed in Argon [36]. Implementation details for some of these schedulers were provided in the previous section. We also compare against the raw device I/O in which requests are issued to the storage devices as soon as they are passed from the file system.

We explain our fairness and efficiency metrics in evaluation. Fairness is defined as the case that each task gains equal access to resources. In a concurrent execution with n tasks, this can be observed if each task experiences a factor of n slowdown compared to running-alone. We call this *proportional slowdown*. Note that better performance may be achieved when some tasks only contain partial I/O load (*i.e.*, they do not make I/O requests for significant parts of their execution). Some tasks may also gain better performance if they are able to utilize the allotted resources more efficiently (*e.g.*, through exploiting device internal parallelism). However, fairness dictates that none should exhibit substantially worse performance than the proportional slowdown.

We also devise a metric to represent the overall system efficiency of a concurrent execution. This metric, we call *concurrent efficiency*, measures the relative throughput of the concurrent execution to the running-alone throughput of individual tasks. Intuitively, it assigns a base efficiency of 1.0 to each task's running-alone performance (at the absence of resource competition and interference) and then weighs the throughput of a concurrent execution against the base efficiency. Consider n concurrent tasks $t_1, t_2, \cdots, t_n$. Let t_i's running-alone throughput be $\text{Thrput}_i^{\text{alone}}$. Let t_i's throughput in the concurrent execution be $\text{Thrput}_i^{\text{conc}}$. Then formally for the concurrent execution:

$$\text{Concurrent efficiency} = \sum_{i=1}^{n} \frac{\text{Thrput}_i^{\text{conc}}}{\text{Thrput}_i^{\text{alone}}}. \qquad (2)$$

An efficiency of less than 1.0 indicates the overhead of concurrent execution or the lack of full utilization of resources. An efficiency of greater than 1.0 indicates the additional benefit of concurrent execution, *e.g.*, due to exploiting the parallelism in the storage device.

Our experiments utilize the Flash-based storage devices described in the beginning of Section 3. They include three (Intel/Mtron/Vertex) Flash-based SSDs as well as a low-power SanDisk CompactFlash drive.

Section 6.1 will first evaluate the fairness and efficiency using a set of synthetic benchmarks with varying I/O concurrency. Section 6.2 then provides evaluation with realistic applications of the SPECweb workload on an Apache web server and the TPC-C workload on a MySQL database. Finally, Section 6.3 performs evaluation on a CompactFlash drive in a low-power wimpy node using the FAWN Data Store workload [2].

6.1 Evaluation with Synthetic I/O Benchmarks

Synthetic I/O benchmarks allow us to flexibly vary parameters in the resource competition. Each synthetic benchmark contains a number of tasks issuing I/O requests of different types and sizes. Evaluation here considers four benchmark cases:

- *1-reader 1-writer* that concurrently runs a reader continuously issuing 4 KB reads and a writer continuously issuing 4 KB writes;
- *4-reader 4-writer* that concurrently runs four 4 KB readers and four 4 KB writers;
- *4-reader 4-writer (with thinktime)* that is like the above case but each task also induces some exponentially distributed thinktime between I/O such that the total thinktime time is approximately equal to its I/O device usage time;
- *4 KB-reader and 128 KB-reader* that concurrently runs a reader continuously issuing 4 KB reads and another reader continuously issuing 128 KB reads.

The last case helps evaluate the value of FIOS for read-only workloads or workloads in which writes are asynchronous and delayed to the background.

Fairness Figure 5 illustrates the fairness and performance of the three read/write benchmark cases under different I/O schedulers. On the two drives (Intel/Mtron SSDs) with strong read/write interference, the raw device I/O, Linux CFQ, and SFQ(D) fail to achieve fairness.

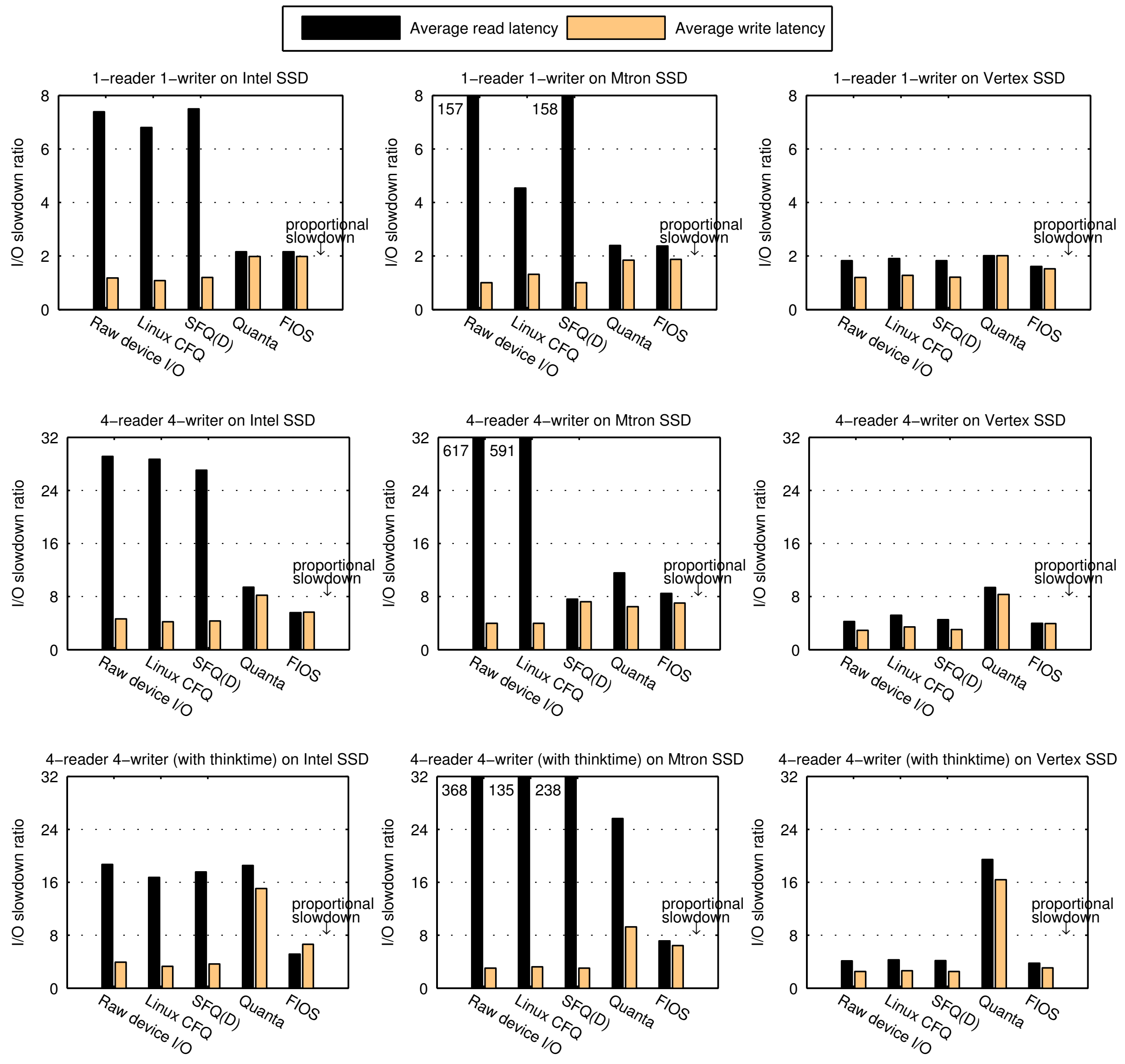


Figure 5: Fairness and performance of synthetic read/write benchmarks under different I/O schedulers. The *I/O slowdown ratio* for read (or write) is the I/O latency normalized to that when running alone. Results cover three Flash-based SSDs (corresponding to the three columns) and three workload scenarios with varying reader/writer concurrency (corresponding to the three rows). For each case, we mark the slowdown ratio that is proportional to the total number of tasks in the system, which is a measure of fairness.

Specifically, readers experience many times the proportional slowdown while writers are virtually unaffected. Because raw device I/O makes no attempt to schedule I/O, reads and writes are interleaved as they are issued by applications, severely affecting the response of read requests. The Linux CFQ does not perform much better because it disables I/O anticipation for non-rotating storage devices like Flash and it suppresses I/O parallelism between concurrent tasks. SFQ(D) also suffers from poor fairness due to its lack of I/O anticipation. For instance, without anticipation, two-task executions degenerate to one-read/one-write interleaved I/O issuance and poor fairness. The quanta scheduler achieves better fairness than other alternatives due to its aggressive maintenance of per-task quantum. However, it suffers from the cost of excessive I/O anticipation and suppression of I/O parallelism. In contrast, FIOS maintains fairness (approximately at or below proportional slowdown) in all the evaluation cases due to our proposed techniques.

On the Vertex SSD, most schedulers achieve good fair-

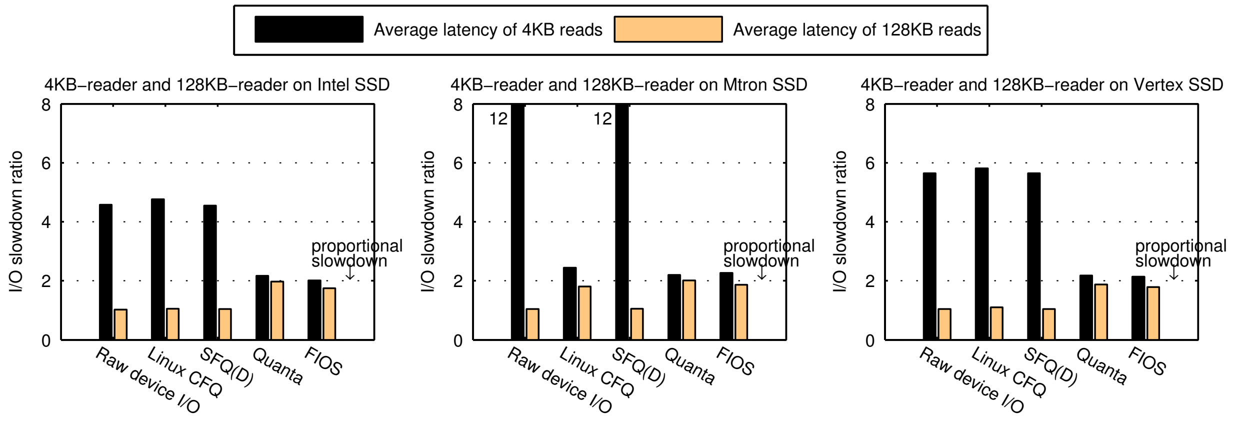


Figure 6: Fairness and performance of two-reader (at different read sizes) benchmark under different I/O schedulers.

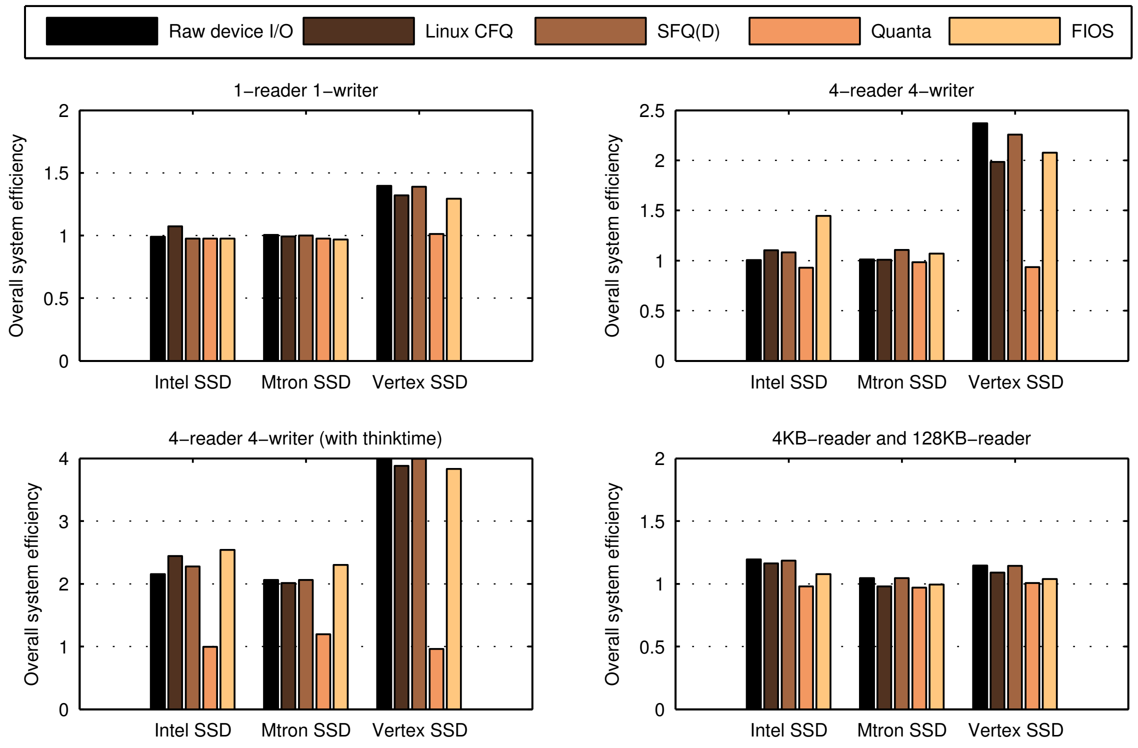


Figure 7: Overall system efficiency of synthetic I/O benchmarks under different I/O schedulers. We use the metric of concurrent efficiency defined in Equation 2. Results cover four benchmark cases and three SSDs.

ness for the read/write benchmark cases due to its modest read/write interference. However, the quanta scheduler still exhibits high cost of excessive I/O anticipation.

Figure 6 shows the fairness and performance of the 4 KB-reader and 128 KB-reader benchmark under different I/O schedulers. Results show that only FIOS and quanta schedulers can maintain fairness in this case. The benefit manifests on all three drives including the Vertex SSD.

Efficiency We next evaluate the overall system efficiency. Figure 7 illustrates the concurrent efficiency (defined in Equation 2) under different I/O schedulers. Results show FIOS achieves higher efficiency when devices allow substantial internal parallelism. These particularly include the two cases with four readers on the Intel and Vertex SSDs. The quanta scheduler exhibits the worst efficiency. This is because its aggressive fairness measures lead to substantial efficiency loss.

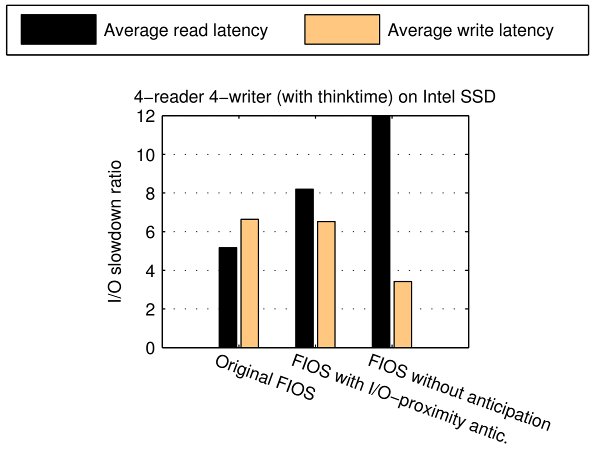


Figure 8: Evaluation on the effect of fairness-oriented I/O anticipation in FIOS on the Intel SSD.

I/O Anticipation for Fairness Figure 8 individually evaluates the effect of fairness-oriented I/O anticipation in FIOS. We compare with two alternatives—no anticipation and anticipation for I/O proximity (as designed in [17] and implemented in Linux). We use the 4-reader 4-writer with thinktime to demonstrate the effect of I/O anticipation. When there is no anticipation, reads suffer substantial additional latency because the deceptive idleness sometimes breaks read preference. While some degree of I/O anticipation is necessary, the conventional I/O anticipation for I/O proximity leads to high performance cost due to excessive idling. The I/O anticipation in FIOS achieves fairness at modest performance cost.

Summary of Results FIOS exhibits better fairness than all alternative schedulers. In terms of efficiency, it is competitive with the best of alternative schedulers in all cases. It is particularly efficient on the Intel SSD because it can exploit its parallelism while managing the read-blocked-by-write problem at the same time.

Among the alternative schedulers, the quanta scheduler is most fair but very inefficient in many cases due to the high cost of its aggressive I/O anticipation. The raw device I/O is most efficient but it is unfair in many situations, particularly in penalizing the reads.

FIOS is not only effective for maintaining fairness between reads and synchronous writes, it is also beneficial for regulating read tasks with different I/O costs. This demonstrates the value of FIOS to support read-only workloads and workloads in which writes are asynchronous and delayed to the background. Further, this makes FIOS valuable for the Vertex drive even though its read/write performance discrepancy is small.

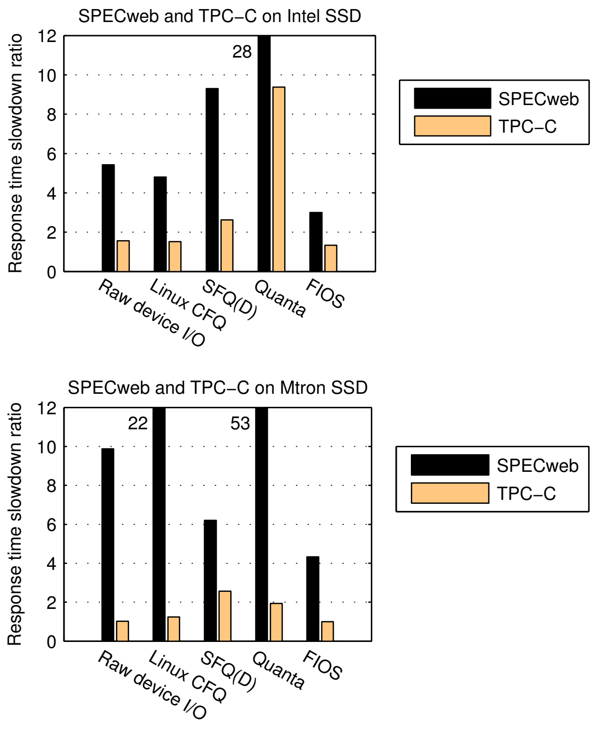


Figure 9: Fairness and performance of SPECweb running with TPC-C under different I/O schedulers. The slowdown ratio for an application is the average request response time normalized to that when the application runs alone. Results cover two Flash-based SSDs.

6.2 Evaluation with SPECweb and TPC-C

Beyond the synthetic benchmarks, we also perform evaluation with realistic workloads. We run the read-only SPECweb99 workload (running on an Apache 2.2.3 web server) along with the write-intensive TPC-C (running on a MySQL 5.5.13 database). Each application is driven by a closed-loop load generator that contains four concurrent clients, each of which issues requests continuously (issuing a new request right after the outstanding one receives a response). The load generators run on a different machine and send requests through the network. This evaluation employs the two drives (Intel/Mtron SSDs) that exhibit large read/write interference effects.

Figure 9 illustrates the fairness and performance results under different I/O schedulers. Unsurprisingly, the read-only SPECweb tends to experience more slowdown than the write-intensive TPC-C does on Flash storage. Among all scheduling approaches, the quanta scheduler exhibits the worst performance and fairness. This is due to its excessive I/O anticipation. Realistic application workloads (like SPECweb and TPC-C) perform significant computation and networking between storage I/O

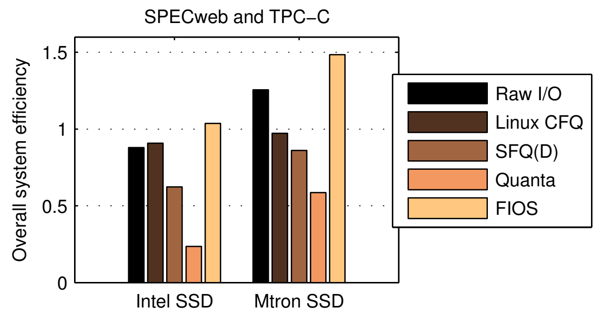


Figure 10: Overall system efficiency of SPECweb running with TPC-C under different I/O schedulers. We use the metric of concurrent efficiency defined in Equation 2. Results cover two Flash-based SSDs.

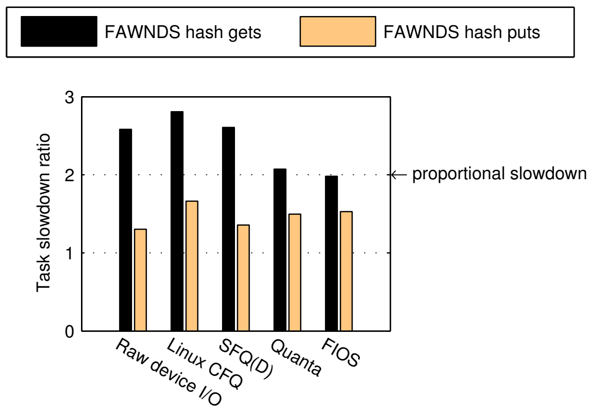


Figure 11: Performance of concurrent FAWN Data Store hash gets (data reads) and hash puts (data writes) on a low-power CompactFlash. The slowdown ratio for a task is defined as its running-alone throughput divided by its throughput at the concurrent run. Higher slowdown ratio means worst performance.

that appears as inter-I/O thinktime. Idling the storage device through such thinktime (as in the quanta scheduler) leads to excessive waste. On the other hand, the poor fairness of the raw device I/O, Linux CFQ, and SFQ(D) is due to a lack of I/O anticipation and poor management of read/write interference on Flash.

FIOS exhibits better fairness and performance than all the alternative approaches, and its performance is more stable across the two SSDs. We measure the fairness as the worst-case application slowdown in a concurrent execution (SPECweb slowdown in all cases). Compared to the quanta scheduler, FIOS reduces the worst-case slowdown by a factor of nine or more on both SSDs. Compared to the raw device I/O, FIOS reduces the worst-case slowdown by a factor of 2.3$\times$ on the Mtron SSD. Compared to the Linux CFQ, FIOS reduces the worst-case slowdown by a factor of five on the Mtron SSD. Compared to SFQ(D), FIOS reduces the worst-case slowdown by about 3.1$\times$ on the Intel SSD.

Figure 10 shows the overall system efficiency of SPECweb running with TPC-C under different I/O schedulers. Results show that FIOS improves the efficiency above the best alternative scheduler by 14% and 18% on the Intel and Mtron SSDs respectively. FIOS achieves high efficiency due to its proper management of read/write interference, I/O parallelism, and controlled I/O anticipation.

6.3 Evaluation on Low-Power CompactFlash

We also test FIOS on a low-power wimpy node like the ones used in the FAWN work [2]. Specifically, the node contains an Alix board with a single-core 500 MHz AMD Geode CPU, 256 MB SDRAM memory, and a 16 GB SanDisk CompactFlash drive. The full node consumes about 5.9 Watts of power at peak load. The CompactFlash, while also NAND Flash-based, is significantly less sophisticated than solid state drives. CompactFlash cards lack the sophisticated firmware and degree of parallelism available in solid-state drives. Despite these differences, CompactFlash still exhibits some of the intrinsic Flash characteristics that FIOS is designed to consider and exploit.

We requested and acquired the FAWN Data Store application from the authors [2]. In our experiments, we concurrently run two FAWN Data Store tasks, one performing hash gets (data reads) and the other performing hash puts (data writes). We run hash puts synchronously to ensure that the data is made persistent before its result is externalized to client. These tasks run against data stores of 1 million records.

Figure 11 presents the resulting get/put slowdown ratios under different I/O schedulers. Only FIOS keeps both hash gets and puts below the proportional slowdown. The quanta scheduler also exhibits good fairness because its suppression of parallelism has no harmful effect on the CompactFlash which does not allow any I/O parallelism. Further, the quanta scheduler's excessive I/O anticipation causes little efficiency loss for FAWN Data Store that performs batched I/O with almost no inter-I/O thinktime. Under all other approaches (raw device I/O, Linux CFQ, and SFQ(D)), hash gets experience worse performance degradation than the proportional slowdown, which indicates poor fairness.

7 Conclusion

Flash-based storage devices are capable of alleviating I/O bottlenecks in data-intensive applications. However, the unique performance characteristics of Flash storage must be taken into account in order to fully exploit their superior I/O capabilities while offering fair access to applications. In this paper, we have characterized the performance of several Flash-based storage devices. We observed that during concurrent access, writes can dramatically affect the response time of read requests. We also observed that Flash-based storage exhibits support for some degree of parallel I/O, though the benefit of parallel I/O varies across devices. Further, the lack of seek/rotation overhead eliminates the performance benefit of anticipatory I/O, but proper I/O anticipation is still needed for the purpose of fairness.

Based on these motivations, we designed a new Flash I/O scheduling approach that contains four essential techniques to ensure fairness with high efficiency—fair timeslice management that allows timeslice fragmentation and concurrent request issuance, read/write interference management, I/O parallelism, and I/O anticipation for fairness. We implemented these design principles in a new I/O scheduler for Linux.

We evaluated our I/O scheduler alongside three alternative fairness-oriented I/O schedulers (Linux CFQ [3], SFQ(D) [18], and a quanta-based I/O scheduler similar to that in Argon [36]). Our evaluation uses a variety of synthetic benchmarks and realistic application workloads on several Flash-based storage devices (including a CompactFlash card in a low-power wimpy node). The results expose the shortcomings of existing I/O schedulers while validating our design principles for Flash resource management. In conclusion, this paper makes the case that fairness warrants the first-class concern in Flash I/O scheduling and it is possible to achieve fairness while attaining high efficiency.

While FIOS is primarily motivated by the Flash read/write interference, we also demonstrate that FIOS is beneficial for regulating the resource usage fairness between read tasks with different I/O costs (a task performing small reads runs concurrently with a task performing large reads). This illustrates the value of FIOS to support read-only workloads and workloads in which writes are asynchronous and delayed to the background. Further, FIOS is also valuable for Flash drives that have modest read/write performance discrepancy.

Acknowledgments We thank Amal Fahad (University of Rochester) for setting up the low-power wimpy node for our evaluation. We also thank Zhuan Chen (University of Rochester) for device installation and machine maintenance when both authors were away from the school. Finally, we thank the anonymous FAST reviewers and our shepherd Randal Burns for comments that helped improve this paper.

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.*, pages 57–70, Boston, MA, June 2008.

[2] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *SOSP'09: 22th ACM Symp. on Operating Systems Principles*, pages 1–14, Big Sky, MT, Oct. 2009.

[3] J. Axboe. Linux block IO — present and future. In *Ottawa Linux Symp.*, pages 51–61, Ottawa, Canada, July 2004.

[4] G. Banga, P. Druschel, and J. Mogul. Resource containers: A new facility for resource management in server systems. In *OSDI'99: Third USENIX Symp. on Operating Systems Design and Implementation*, pages 45–58, New Orleans, LA, Feb. 1999.

[5] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Disk scheduling with quality of service quarantees. In *IEEE Int'l Conf. on Multimedia Computing and Systems*, pages 400–405, Florence , Italy, June 1999.

[6] A. M. Caulfield, L. M. Grupp, and S. Swanson. Gordon: Using flash memory to build fast, power-efficient clusters for data-intensive applications. In *ASPLOS'09: 14th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 217–228, Washington, DC, Mar. 2009.

[7] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of Flash memory based solid state drives. In *ACM SIGMETRICS*, pages 181–192, Seattle, WA, June 2009.

[8] F. Chen, R. Lee, and X. Zhang. Essential roles of exploiting internal parallelism of Flash memory based solid state drives in high-speed data processing. In *HPCA'11: 17th IEEE Symp. on High Performance Computer Architecture*, pages 266–277, San Antonio, TX, Feb. 2011.

[9] S. Chen. FlashLogging: Exploiting Flash devices for synchronous logging performance. In *SIGMOD'09: 35th Int'l Conf. on Management of Data*, pages 73–86, Providence, RI, June 2009.

[10] Choi et al. A 20nm 1.8V 8Gb PRAM with 40MB/s program bandwidth. In *ISSCC'12: Int'l Solid-State Circuits Conf.*, San Francisco, CA, Feb. 2012.

[11] H. Dai, M. Neufeld, and R. Han. ELF: An efficient log-structured Flash file system for micro sensor nodes. In *SenSys'04: Second ACM Conf. on Embedded Networked Sensor Systems*, pages 176–187, Baltimore, MD, Nov. 2004.

[12] De Sandre et al. A 4 Mb LV MOS-selected embedded phase change memory in 90 nm standard CMOS technology. *IEEE Journal of Solid-State Circuits*, 46(1):52–63, Jan. 2011.

[13] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *ACM SIGCOMM*, pages 1–12, Austin, TX, Sept. 1989.

[14] M. Dunn and A. L. N. Reddy. A new I/O scheduler for solid state devices. Technical Report TAMU-ECE-2009-02, Dept. of Electrical and Computer Engineering, Texas A&M Univ., Apr. 2009.

[15] P. Goyal, H. M. Vin, and H. Cheng. Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Trans. on Networking*, 5(5):690–704, Oct. 1997.

[16] A. Gulati, A. Merchant, and P. J. Varman. pClock: An arrival curve based approach for QoS guarantees in shared storage systems. In *ACM SIGMETRICS*, pages 13–24, San Diego, CA, June 2007.

[17] S. Iyer and P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *SOSP'01: 18th ACM Symp. on Operating Systems Principles*, pages 117–130, Banff, Canada, Oct. 2001.

[18] W. Jin, J. S. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. In *ACM SIGMETRICS*, pages 37–48, New York, NY, June 2004.

[19] T. Kelly, A. H. Karp, M. Stiegler, T. Close, and H. K. Cho. Output-valid rollback-recovery. Technical Report HPL-2010-155, HP Laboratories, Oct. 2010.

[20] J. Kim, J. M. Kim, S. Noh, S. L. Min, and Y. Cho. A space-efficient Flash translation layer for CompactFlash systems. *IEEE Trans. on Consumer Electronics*, 48(2):366–375, May 2002.

[21] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh. Disk schedulers for solid state drives. In *EMSOFT'09: 7th ACM Conf. on Embedded Software*, pages 295–304, Grenoble, France, Oct. 2009.

[22] J. Kim, S. Seo, D. Jung, J.-S. Kim, and J. Huh. Parameter-aware I/O management for solid state disks (SSDs). *IEEE Trans. on Computers*, Apr. 2011.

[23] 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, July 2006.

[24] J. Lee, S. Kim, H. Kwon, C. Hyun, S. Ahn, J. Choi, D. Lee, and S. H. Noh. Block recycling schemes and their cost-based optimization in NAND Flash memory based storage system. In *EMSOFT'07: 7th ACM Conf. on Embedded Software*, pages 174–182, Salzburg, Austria, Oct. 2007.

[25] A. Leventhal. Flash storage memory. *Communications of the ACM*, 51(7):47–51, July 2008.

[26] P. Lu and K. Shen. Virtual machine memory access tracing with hypervisor exclusive cache. In *USENIX Annual Technical Conf.*, pages 29–43, Santa Clara, CA, June 2007.

[27] C. R. Lumb, A. Merchant, and G. A. Alvarez. Facade: Virtual storage devices with performance guarantees. In *FAST'03: Second USENIX Conf. on File and Storage Technologies*, pages 131–144, San Francisco, CA, Apr. 2003.

[28] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the sync. In *OSDI'06: 7th USENIX Symp. on Operating Systems Design and Implementation*, Seattle, WA, Nov. 2006.

[29] A. K. Parekh. *A generalized processor sharing approach to flow control in integrated services networks*. PhD thesis, Dept. Elec. Eng. Comput. Sci., MIT, 1992.

[30] S. Park and K. Shen. A performance evaluation of scientific I/O workloads on flash-based SSDs. In *IASDS'09: Workshop on Interfaces and Architectures for Scientific Data Storage*, New Orleans, LA, Sept. 2009.

[31] M. Polte, J. Simsa, and G. Gibson. Comparing performance of solid state devices and mechanical disks. In *3rd Petascale Data Storage Workshop*, Austin, TX, Nov. 2008.

[32] A. Povzner, T. Kaldewey, S. Brandt, R. Golding, T. M. Wong, and C. Maltzahn. Efficient guaranteed disk request scheduling with Fahrrad. In *EuroSys'08: Third ACM European Conf. on Computer Systems*, pages 13–25, Glasgow, Scotland, Apr. 2008.

[33] A. L. N. Reddy, J. Wyllie, and K. B. R. Wijayaratne. Disk scheduling in a multimedia I/O system. *ACM Trans. on Multimedia Computing, Communications, and Applications*, 1(1):37–59, Feb. 2005.

[34] K. Shen. Request behavior variations. In *ASPLOS'10: 15th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 103–116, Pittsburg, PA, Mar. 2010.

[35] K. Shen, C. Stewart, C. Li, and X. Li. Reference-driven performance anomaly identification. In *ACM SIGMETRICS*, pages 85–96, Seattle, WA, June 2009.

[36] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger. Argon: Performance insulation for shared storage servers. In *FAST'07: 5th USENIX Conf. on File and Storage Technologies*, pages 61–76, San Jose, CA, Feb. 2007.

[37] J. Zhang, A. Sivasubramaniam, Q. Wang, A. Riska, and E. Riedel. Storage performance virtualization via throughput and latency control. *ACM Trans. on Storage*, 2(3):283–308, Aug. 2006.

Shredder: GPU-Accelerated Incremental Storage and Computation

Pramod Bhatotia[†] *Rodrigo Rodrigues*[†] *Akshat Verma*[‡]

[†]*Max Planck Institute for Software Systems (MPI-SWS) and* [‡]*IBM Research – India*

Abstract

Redundancy elimination using data deduplication and incremental data processing has emerged as an important technique to minimize storage and computation requirements in data center computing. In this paper, we present the design, implementation and evaluation of Shredder, a high performance content-based chunking framework for supporting incremental storage and computation systems. Shredder exploits the massively parallel processing power of GPUs to overcome the CPU bottlenecks of content-based chunking in a cost-effective manner. Unlike previous uses of GPUs, which have focused on applications where computation costs are dominant, Shredder is designed to operate in both compute-and data-intensive environments. To allow this, Shredder provides several novel optimizations aimed at reducing the cost of transferring data between host (CPU) and GPU, fully utilizing the multicore architecture at the host, and reducing GPU memory access latencies. With our optimizations, Shredder achieves a speedup of over 5*X* for chunking bandwidth compared to our optimized parallel implementation without a GPU on the same host system. Furthermore, we present two real world applications of Shredder: an extension to HDFS, which serves as a basis for incremental MapReduce computations, and an incremental cloud backup system. In both contexts, Shredder detects redundancies in the input data across successive runs, leading to significant savings in storage, computation, and end-to-end completion times.

1 Introduction

With the growth in popularity of Internet services, online data stored in data centers is increasing at an ever-growing pace. In 2010 alone, mankind is estimated to have produced $1,200$ exabytes of data [1]. As a result of this "data deluge," managing storage and computation over this data has become one of the most challenging tasks in data center computing.

A key observation that allows us to address this challenge is that a large fraction of the data that is produced and the computations performed over this data are redundant; hence, *not* storing redundant data or performing redundant computation can lead to significant savings in terms of both storage and computational resources. To make use of redundancy elimination, there exist a series of research and product proposals (detailed in §8) for performing *data deduplication* and *incremental computations*, which avoid storing or computing tasks based on redundant data, respectively.

Both data deduplication schemes and incremental computations rely on storage systems to detect duplicate content. In particular, the most effective way to perform this detection is using *content-based chunking*, a technique that was pioneered in the context of the LBFS [33] file system, where chunk boundaries within a file are dictated by the presence of certain content instead of a fixed offset. Even though content-based chunking is useful, it is a computationally demanding task. Chunking methods need to scan the entire file contents, computing a fingerprint over a sliding window of the data. This high computational cost has caused some systems to simplify the fingerprinting scheme by employing sampling techniques, which can lead to missed opportunities for eliminating redundancies [9]. In other cases, systems skip content-based chunking entirely, thus forgoing the opportunity to reuse identical content in similar, but not identical files [22]. Therefore, as we get flooded with increasing amounts of data, addressing this computational bottleneck becomes a pressing issue in the design of storage systems for data center-scale systems.

To address this issue we propose Shredder, a system for performing efficient content-based chunking to support scalable incremental storage and computations. Shredder builds on the observation that neither the exclusive use of multicore CPUs nor the specialized hardware accelerators is sufficient to deal with large-scale data in a cost-effective manner: multicore CPUs alone cannot sustain a high throughput, whereas the specialized hardware accelerators lack programmability for other tasks and are costly. As an alternative, we explore employing modern GPUs to meet these high computational requirements (while, as evidenced by prior research [23, 26], also allowing for a low operational cost). The application of GPUs in this setting, however, raises a significant challenge — while GPUs have shown to produce performance improvements for computation intensive applications, where CPU dominates the overall cost envelope [23, 24, 26, 43, 44], it still remains to be proven that GPUs are equally as effective for data intensive applications, which need to perform large data transfers for a significantly smaller amount of processing.

To make the use of GPUs effective in the context of storage systems, we designed several novel techniques, which we apply to two proof-of-concept applications. In particular, Shredder makes the following technical contributions:

GPU acceleration framework. We identified three key challenges in using GPUs for data intensive applications, and addressed them with the following techniques:

- **Asynchronous execution.** To minimize the cost of transferring data between host (CPU) and GPU, we use a double buffering scheme. This enables GPUs to perform computations while simultaneously data is transferred in the background. To support this background data transfer, we also introduce a ring buffer of pinned memory regions.
- **Streaming pipeline.** To fully utilize the availability of a multicore architecture at the host, we use a pipelined execution for the different stages of content-based chunking.
- **Memory coalescing.** Finally, because of the high degree of parallelism, memory latencies in the GPU will be high due to the presence of random access across multiple bank rows of GPU memory, which leads to a higher number of conflicts. We address this problem with a cooperative memory access scheme, which reduces the number of fetch requests and bank conflicts.

Use cases. We present two applications of Shredder to accelerate storage systems. The first case study is a system called Inc-HDFS, a file-system that is based on HDFS but is designed to support incremental computations for MapReduce jobs. Inc-HDFS leverages Shredder to provide a mechanism for identifying similarities in the input data of consecutive runs of the same MapReduce job. In this way Inc-HDFS enables efficient incremental computation, where only the tasks whose inputs have changed need to be recomputed. The second case study is a backup architecture for a cloud environment, where VMs are periodically backed up. We use Shredder on a backup server and use content-based chunking to perform efficient deduplication and significantly improve backup bandwidth.

We present experimental results that establish the effectiveness of the individual techniques we propose, as well as the ability of Shredder to improve the performance of the two real-world storage systems.

The rest of the paper is organized as follows. In Section 2, we provide background on content-based chunking, and discuss specific architectural features of GPUs. An overview of the GPU acceleration framework and its scalability challenges are covered in Section 3. Section 4 presents present a detailed system design, namely several performance optimizations for increasing Shredder's throughput. We present the implementation and evaluation of Shredder in Section 5. We cover the two case studies in Section 6 and Section 7. Finally, we discuss related work in Section 8, and conclude in Section 9.

2 Background

In this section, we first present background on content-based chunking, to explain its cost and potential for parallelization. We then provide a brief overview of the massively parallel compute architecture of GPUs, namely their memory subsystem and its limitations.

2.1 Content-based Chunking

Identification of duplicate data blocks has been used for deduplication systems in the context of both storage [33, 39] and incremental computation frameworks [14]. For storage systems, the duplicate data blocks need not to be stored and, in the case of incremental computations, a sub-computation based on the duplicate content may be reused. Duplicate identification essentially consists of:

1. Chunking: This is the process of dividing the data set into chunks in a way that aids in the detection of duplicate data.
2. Hashing: This is the process of computing a collision-resistant hash of the chunk.
3. Matching: This is the process of checking if the hash for a chunk already exists in the index. If it exists then there is a duplicate chunk, else the chunk is new and its hash is added to the index.

This paper focuses on the design of chunking schemes (step 1), since this can be, in practice, one of the main bottlenecks of a system that tries to perform this class of optimizations [9, 22]. Thus we begin by giving some background on how chunking is performed.

One of the most popular approaches for content-based chunking is to compute a Rabin fingerprint [40] over sliding windows of w contiguous bytes. The hash values produced by the fingerprinting scheme are used to create chunk boundaries by starting new chunks whenever the computed hash matches one of a set of markers (e.g., its value $mod\ p$ is lower or equal to a constant). In more detail, given a w-bit sequence, it is represented as a polynomial of degree $w-1$ over the finite field $GF(2)$:

$$f(x) = m_0 + m_1 x + \cdots + m_{w-1} x^{w-1} \quad (1)$$

Given this polynomial, an irreducible polynomial $div(x)$ of degree k is chosen. The fingerprint of the original bit sequence is the remainder $r(x)$ obtained by division of $f(x)$ using $div(x)$. Chunk boundary is defined when the fingerprint takes some pre-defined specific values called markers. In addition, practical schemes define a minimum *min* and maximum *max* chunk size, which implies that after finding a marker the fingerprint computation can skip *min* bytes, and that a marker is always set when a total of *max* bytes (including the skipped portion) have been scanned without finding a marker. The minimum size limits the metadata overhead for index management and the maximum size limits the size of the RAM buffers that are required. Throughout the rest of the paper, we will use $min = 0$ and $max = \infty$ unless otherwise noted.

Rabin fingerprinting is computationally very expensive. To minimize the computation cost, there has been work on reducing chunking time by using sampling techniques, where only a subset of bytes are used for chunk identification (e.g., SampleByte [9]). However, such approaches are limiting because they are suited only for small sized chunks, as skipping a large number of bytes leads to missed opportunities for deduplication. Thus, Rabin fingerprinting still remains one of the most popular chunking schemes, and reducing its computational cost presents a fundamental challenge for improving systems that make use of duplicate identification.

When minimum and maximum chunk sizes are not required, chunking can be parallelized in a way that different threads operate on different parts of the data completely independent of each other, with the exception of a small overlap of the size of the sliding window (w bytes) near partition boundaries. Using *min* and *max* chunk sizes complicates this task, though schemes exist to achieve efficient parallelization in this setting [29, 31].

2.2 General-Purpose Computing on GPUs

GPU architecture. GPUs are highly parallel, multi-threaded, many-core processors with tremendous computational power and very high memory bandwidth. The high computational power is derived from the specialized design of GPUs, where more transistors are devoted to simple data processing units (ALUs) rather than used to integrate sophisticated pre-fetchers, control flows and data caches. Hence, GPUs are well-suited for data-parallel computations with high arithmetic intensity rather than data caching and flow control.

Figure 1 illustrates a simplified architecture of a GPU. A GPU can be modeled as a set of Streaming Multiprocessors (SMs), each consisting of a set of scalar processor cores (SPs). An SM works as SIMT (Single Instruction, Multiple Threads), where the SPs of a multiprocessor execute the same instruction simultaneously but on different data elements. The data memory in the GPU is organized as multiple hierarchical spaces for threads in execution. The GPU has a large high-bandwidth device memory with high latency. Each SM also contains a very fast, low latency on-chip shared memory to be shared among its SPs. Also, each thread has access to a private local memory.

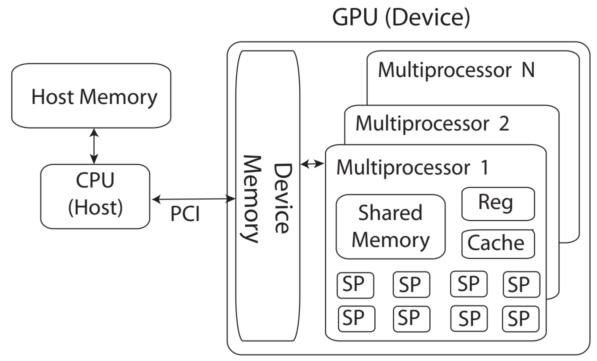


Figure 1: A simplified view of the GPU architecture.

Overall, a GPU architecture differs from a traditional processor architecture in the following ways: (*i*) an order of magnitude higher number of arithmetic units; (*ii*) minimal support for prefetching and buffers for outstanding instructions; (*iii*) high memory access latencies and higher memory bandwidth.

Programming model. The CUDA [6] programming model is amongst the most popular programming models to extract parallelism and scale applications on GPUs. In this programming model, a host program runs on the CPU and launches a kernel program to be executed on the GPU device in parallel. The kernel executes as a grid of one or more thread blocks, each of which is dynamically scheduled to be executed on a single SM. Each thread block consists of a group of threads that cooperate with each other by synchronizing their execution and sharing multiprocessor resources such as shared memory and registers. Threads within a thread block get executed on a multiprocessor in scheduling units of 32 threads, called a warp. A half-warp is either the first or second half of a warp.

2.3 SDRAM Access Model

Offloading chunking to the GPU requires a large amount of data to be transferred from the host to the GPU memory. Thus, we need to understand the performance of the memory subsystem in the GPU, since it is critical to chunking performance.

The global memory in the Nvidia C2050 GPU is GDDR5, which is based on the DDR3 memory architecture [2]. Memory is arranged into banks and banks are organized into rows. Every bank also has a sense amplifier, into which a row must be loaded before any data from the row can be read by the GPU. Whenever a memory location is accessed, an *ACT* command selects the corresponding bank and brings the row containing the memory location into a sense amplifier. The appropriate word is then transferred from the sense amplifier. When an access to a second memory location is performed within the same row, the data is transferred directly from the sense amplifier. On the other hand, if the data is accessed from a different row in the bank, a *PRE* (pre-charge) command writes the previous data back from the sense amplifier to the memory row. A second *ACT* command is performed to bring the row into the sense amplifier.

Note that both *ACT* and *PRE* commands are high latency operations that contribute significantly to overall memory latency. If multiple threads access data from different rows of the same bank in parallel, that sense amplifier is continually activated (*ACT*) and pre-charged (*PRE*) with different rows, leading to a phenomenon called bank conflict. In particular, a high degree of uncoordinated parallel access to the memory subsystem is likely to result in a large number of bank conflicts.

3 System Overview and Challenges

In this section, we first present the basic design of Shredder. Next, we explain the main challenges in scaling up our basic design.

3.1 Basic GPU-Accelerated Framework

Figure 2 depicts the workflow of the basic design for the Shredder chunking service. In this initial design, a multi-threaded program running in user mode on the host (i.e., on the CPU) drives the GPU-based computations. The framework is composed of four major modules. First, the *Reader* thread on the host receives the data stream (e.g., from a SAN), and places it in the memory of the host for content-based chunking. After that, the *Transfer* thread allocates global memory on the GPU and uses the DMA controller to transfer input data from the host memory to the allocated GPU (device) memory. Once the data transfer from the CPU to the GPU is complete, the host launches the *Chunking* kernel for parallel sliding window computations on the GPU. Once the chunking kernel finds all resulting chunk boundaries for the input data, the *Store* thread transfers the resulting chunk boundaries from the device memory to the host memory. When minimum and maximum chunk sizes are set, the *Store* thread also adjusts the chunk set accordingly. Thereafter, the *Store* thread uses an upcall to notify the chunk boundaries to the application that is using the Shredder library.

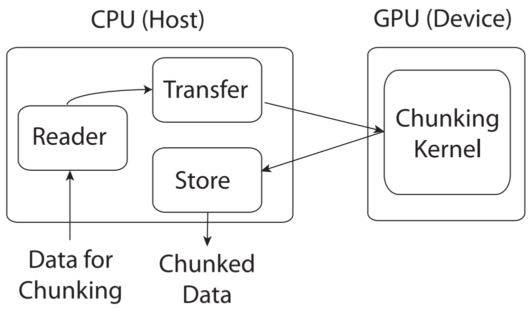


Figure 2: Basic workflow of Shredder.

The chunking kernel is responsible for performing parallel content-based chunking of the data present in the global memory of the GPU. Accesses to the data are performed by multiple threads that are created on the GPU by launching the chunking kernel. The data in the GPU memory is divided into equal sized sub-streams, as many as the number of threads. Each thread is responsible for handling one of these sub-streams. For each sub-stream, a thread computes a Rabin fingerprint in a sliding window manner. In particular, each thread examines a 48-byte region from its assigned sub-stream, and computes the Rabin fingerprint for the selected region. The thread compares the resulting low-order 13 bits of the region's fingerprint with a pre-defined marker. This leads to an expected chunk size of 4 KB. If the fingerprint matches the marker then the thread defines that particular region as the end of a chunk boundary. The thread continues to compute the Rabin fingerprint in a sliding window manner in search of new chunk boundaries by shifting a byte forward in the sub-stream, and repeating this process.

3.2 Scalability Challenges

The basic design for Shredder that we presented in the previous section corresponds to the traditional way in which GPU-assisted applications are implemented. This design has proven to be sufficient for computation-intensive applications, where the computation costs can dwarf the cost of transferring the data to the GPU memory and accessing that memory from the GPU's cores. However, it results in only modest performance gains for data intensive applications that perform single-pass processing over large amounts of data, with a computational cost that is significantly lower than traditional GPU-assisted applications.

To understand why this is the case, we present in Table 1 some key performance characteristics of a specific GPU architecture (NVidia Tesla C2050), which helps us explain some important bottlenecks for GPU-accelerated applications. In particular, and as we will demonstrate in subsequent sections, we identified the following bottle-

Parameter	Value
GPU Processing Capacity	1030 GFlops
Reader (I/O) Bandwidth	2 GBps
Host-to-Device Bandwidth	5.406 GBps
Device-to-Host Bandwidth	5.129 GBps
Device Memory Latency	400 - 600 cycles
Device Memory Bandwidth	144 GBps
Shared Memory Latency	L1 latency (a few cycles)

Table 1: Performance characteristics of the GPU (NVidia Tesla C2050)

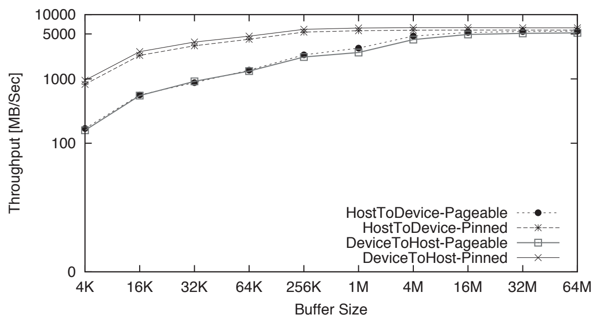


Figure 3: Bandwidth test between host and device.

necks in the basic design of Shredder.

GPU device memory bottleneck. The fact that data needs to be transferred to the GPU memory before being processed by the GPU represents a serial dependency: such processing only starts to execute after the corresponding transfer concludes.

Host bottleneck. The host machine performs three serialized steps (performed by the Reader, Transfer, and Store threads) in each iteration. Since these three steps are inherently dependent on each other for a given input buffer, this serial execution becomes a bottleneck at host. Also, given the availability of multicore architecture at the host, this serialized execution leads to an underutilization of resources at host.

High memory latencies and bank conflicts. The global device memory on the GPU has a high latency, of the order of 400 to 600 cycles. This works well for HPC algorithms, which are quadratic $O(N^2)$ or a higher degree polynomial in the input size N, since the computation time hides the memory access latencies. Chunking is also compute intensive, but it is only linear in the input size ($O(N)$, though the constants are high). Hence, even though the problem is compute intensive on traditional CPUs, on a GPU with an order of magnitude larger number of scalar cores, the problem becomes memory-intensive. In particular, the less sophisticated memory subsystem of the GPU (without prefetching or data caching support) is stressed by frequent memory access by a massive number of threads in parallel. Furthermore, a higher degree of parallelism causes memory to be accessed randomly across multiple bank rows, and leads to a very high number of bank conflicts. As a result, it becomes difficult to hide the latencies of accesses to the device memory.

4 Shredder Optimizations

In this section, we describe several novel optimizations that extend the basic design to overcome the challenges we highlighted in the previous section.

4.1 Device Memory Bottlenecks

4.1.1 Concurrent Copy and Execution

The main challenge we need to overcome is the fact that traditional GPU-assisted applications that follow the basic design were designed for a scenario where the cost of transferring data to the GPU is significantly outweighed by the actual computation cost. In particular, the basic design serializes the execution of copying data to the GPU memory and consuming the data from that memory by the Kernel thread. This serialized execution may not suit the needs of data intensive applications, where the cost of the data transfer step becomes a more significant fraction of the overall computation time.

To understand the magnitude of this problem, we measured the overhead of a DMA transfer of data between the host and the device memory over the PCIe link connected to GPU. Figure 3 summarizes the effective bandwidth between host memory and device memory for different buffer sizes. We measured the bandwidth both ways between the host and the device to gauge the DMA overhead for the Transfer and the Store thread. Note that the effective bandwidth is a property of the DMA controller and the PCI bus, and it is independent of the number of threads launched in the GPU. In this experiment, we also varied the buffer type allocated for the host memory region, which is allocated either as pageable or pinned memory regions. (The need for pinned memory will become apparent shortly.)

Highlights. Our measurements demonstrate the following: (i) small sized buffer transfers are more expensive than those using large sized buffers; (ii) the throughput saturates for buffer sizes larger than 32 MB (for pageable memory region) and 256 KB (for pinned memory region); (iii) for large sized buffers (greater than 32 MB), the throughput difference between pageable and pinned memory regions is not significant; and (iv) the effective bandwidth of the PCIe bus for data transfer is on the order of 5 GB/sec, whereas the global device memory access time by scalar processors in GPUs is on the order of

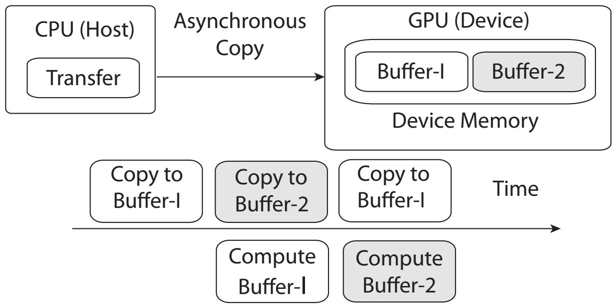


Figure 4: Concurrent copy and execution.

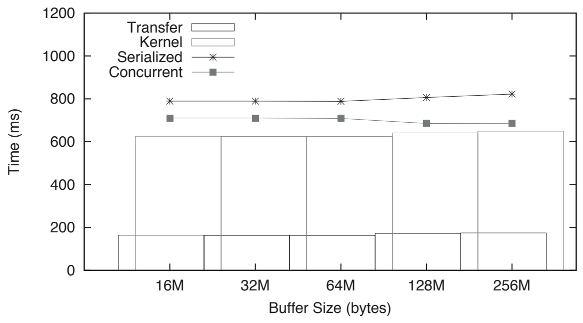


Figure 5: Normalized overlap time of communication with computation with varied buffer sizes for 1GB data.

144 GB/sec, an order of magnitude higher.

Implications. The time spent to chunk a given buffer is split between the memory transfer and the kernel computation. For a non-optimized implementation of the chunking computation, we spend approximately 25% of the time performing the transfer. Once we optimize the processing in the GPU, the host to GPU memory transfer may become an even greater burden on the overall performance.

Optimization. In order to avoid the serialized execution of the copy and data consumption steps, we propose to overlap the copy and the execution phases, thus allowing for the concurrent execution of data communication and the chunking kernel computations. To enable this, we designed a double buffering technique as shown in Figure 4, where we partition the device memory into twin buffers. These twin buffers will be alternatively used for communication and computation. In this scheme, the host asynchronously copies the data into the first buffer and, in the background, the device works on the previously filled second buffer. To be able to support asynchronous communication, the host buffer is allocated as a pinned memory region, which prevents the region from being swapped out by the pager.

Effectiveness. Figure 5 shows the effectiveness of the double buffering approach, where the histogram for transfer and kernel execution shows a 30% time overlap between the concurrent copy and computation. Even though the total time taken for concurrent copy and execution (`Concurrent`) is reduced by only 15% as compared to the serialized execution (`Serialized`), it is important to note that the total time is now dictated solely by the compute time. Hence, double buffering is able to remove the data copying time from the critical path, allowing us to focus only on optimizing the computation time in the GPU (which we address in § 4.3).

To support the concurrent copy and execution, however, requires us to pin memory at the host, which reduces the memory allocation performance at the host. We next present an optimization to handle this side effect and ensure that double buffering leads to an end-to-end increase in chunking bandwidth.

4.1.2 Circular Ring Pinned Memory Buffers

As explained above, the double buffering requires an asynchronous copy between host memory and device memory. To support this asynchronous data transfer, the host side buffer should be allocated as a pinned memory region. This locks the corresponding page so that accessing that region does not result in a page fault until the region is subsequently unpinned.

To quantify the allocation overheads of using a pinned memory region, we compared the time required for dynamic memory allocation (using `malloc`) and pinned memory allocation (using the CUDA memory allocator wrapper). Since Linux follows an optimistic memory allocation strategy, where the actual allocation is deferred until memory initialization, in our measurements we initialized the memory region (using `bzero`) to force the kernel to allocate the desired buffer size. Figure 6 compares the allocation overhead of pageable and pinned memory for different buffer sizes.

Highlights. The important take away points are the following: (i) pinned memory allocation is more expensive than the normal dynamic memory allocation; and (ii) an adverse side effect of having too many pinned memory pages is that it can increase paging activity for unpinned pages, which degrades performance.

Implications. The main implication for our system design is that we need to minimize the allocation of pinned memory region buffers, to avoid increased paging activity or even thrashing.

Optimization. To minimize the allocation of pinned memory region while restricting ourselves to using the CUDA architecture, we designed a circular ring buffer built from a pinned memory region, as shown in Figure 7, with the property that the number of buffers can be kept low (namely as low as the number of stages in the streaming pipeline, as described in §4.2). The pinned

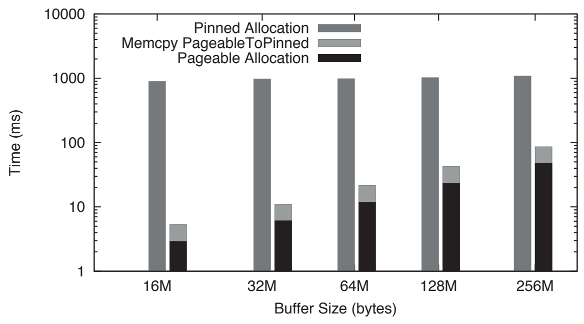


Figure 6: Comparison of allocation overhead of pageable with pinned memory region.

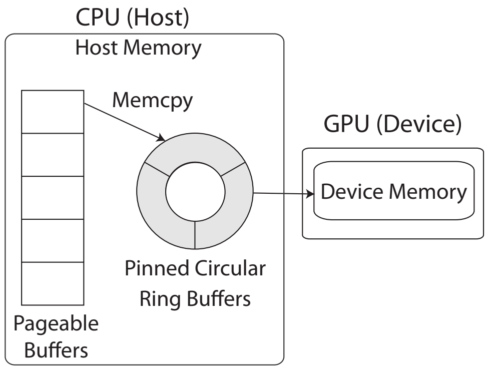


Figure 7: Ring buffer for the pinned memory region.

regions in the circular buffer are allocated only once during the system initialization, and thereafter are reused in a round-robin fashion after the transfer between the host and the device memory is complete. This allows us to keep the overhead of costly memory allocation negligible and have sufficient memory pages for other tasks.

Effectiveness. Figure 6 shows the effectiveness of our approach, where we compare the time for allocating pageable and pinned memory regions. Since we incur the additional cost of copying the data from pageable memory to the pinned memory region, we add this cost to the total cost of using pageable buffers. Overall, our approach is faster by an order of magnitude, which highlights the importance of this optimization.

4.2 Host Bottleneck

The previously stated optimizations alleviate the device memory bottleneck for DMA transfers, and allow the device to focus on performing the actual computation. However, the host side modules can still become a bottleneck due to the serialized execution of the following stages (`Reader`→`Transfer`→`Kernel`→`Store`). In this case, the fact that all four modules are serially executed leads to an underutilization of resources at the host side.

To quantify this underutilization at the host, we measured the number of idle spare cycles per core after the launch of the asynchronous execution of the kernel. Table 2 shows the number of RDTSC tick cycles for different buffer sizes. The RDTSC [8] (Read-Time Stamp Counter) instruction keeps an accurate count of every cycle that occurs on the processor for monitoring the performance. The device execution time captures the asynchronous copy and execution of the kernel, and the host kernel launch time measures the time for the host to launch the asynchronous copy and the chunking kernel.

Highlights. These measurements highlight the following: (i) the kernel launch time is negligible compared to the total execution time for the kernel; (ii) the host is idle during the device execution time; and (ii) the host has a large number of spare cycles per core, even with a small sized buffer.

Buffer size (bytes)	16M	32M	64M	128M	256M
Device execution time (ms)	11.39	22.74	42.85	85.7	171.4
Host kernel launch time (ms)	0.03	0.03	0.03	0.08	0.09
Total execution time (ms)	11.42	22.77	42.88	85.78	171.49
Host `RDTSC` ticks @ 2.67 GHz	3.0e7	6.1e7	1.1e8	2.7e8	5.3e8

Table 2: Host spare cycles per core due to asynchronous data-transfer and kernel launch.

Implications. Given the prevalence of host systems running on multicore architectures, the sequential execution of the various components leads to the underutilization of the host resources, and therefore these resources should be used to perform other operations.

Optimization. To utilize these spare cycles at the host, Shredder makes use of a multi-stage streaming pipeline as shown in Figure 8. The goal of this design is that once the Reader thread finishes writing the data in the host main memory, it immediately proceeds to handling a new window of data in the stream. Similarly, the other threads follow this pipelined execution without waiting for the next stage to finish.

To handle the specific characteristics of our pipeline stages, we use different design strategies for different modules. Since the Reader and Store modules deal with I/O, they are implemented as Asynchronous I/O (as described in §5.2.1), whereas the transfer and kernel threads are implemented using multi-buffering (a generalization of the double buffering scheme described in §4.1.1).

Effectiveness. Figure 9 shows the average speedup from using our streaming pipeline, measured as the ratio of time taken by a sequential execution to the time taken by our multi-stage pipeline. We varied the number of pipeline stages that can be executed simultaneously (by

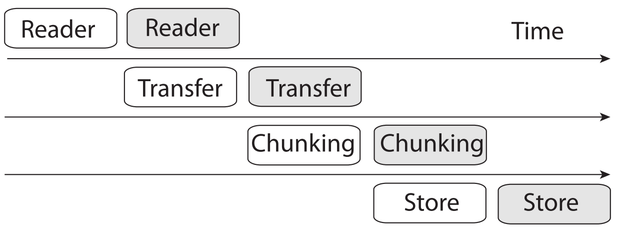


Figure 8: Multi-staged streaming pipeline.

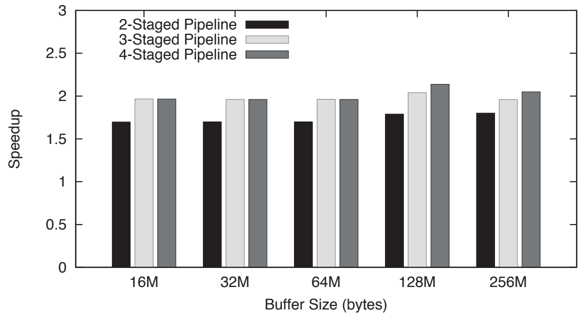


Figure 9: Speedup for streaming pipelined execution.

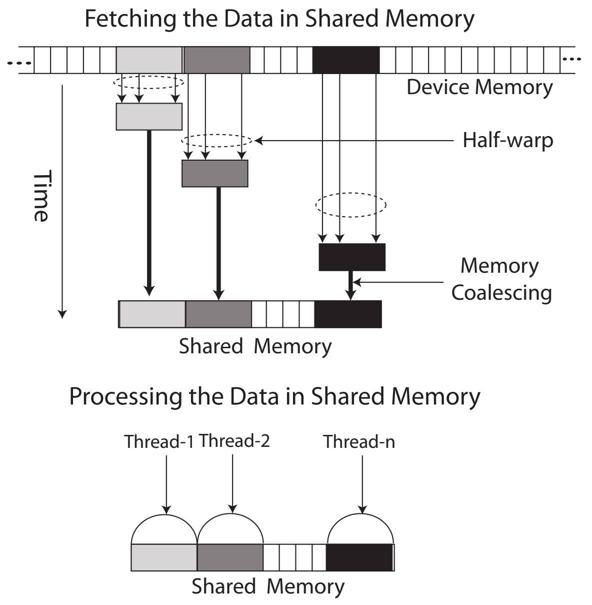


Figure 10: Memory coalescing to fetch data from global device memory to the shared memory.

restricting the number of buffers that are admitted to the pipeline) from 2 to 4. The results show that a full pipeline with all four stages being executed simultaneously achieves a speedup of 2; the reason why this is below the theoretical maximum of a 4X gain is that the various stages do not have equal cost.

4.3 Device Memory Conflicts

We have observed (in Figure 5) that the chunking kernel dominates the overall time spent by the GPU. In this context, it is crucial to try to minimize the contribution of the device memory access latency to the overall cost.

Highlights. The very high access latencies of the device memory (on the order of 400-600 cycles @ 1.15 GHz) and the lack of support for data caching and prefetching can imply a significant overhead in the overall execution time of the chunking kernel.

Implications. The hierarchical memory of GPUs provides us an opportunity to hide the latencies of the global device memory by instead making careful use of the low latency shared memory. (Recall from § 2.2 that the shared memory is a fast and low latency on-chip memory which is shared among a subset of the GPU's scalar processors.) However, fetching data from global to the shared memory requires us to be careful to avoid bank conflicts, which can negatively impact the performance of the GPU memory subsystem. This implies that we should try to improve the inter-thread coordination in fetching data from the device global memory to avoid these bank conflicts.

Optimization. We designed a thread cooperation mechanism to optimize the process of fetching data from the global memory to the shared memory, as shown in Figure 10. In this scheme, a single block that is needed by a given thread is fetched at a time, but each block is fetched with the cooperation of all the threads, and their coordination to avoid bank conflicts. The idea is to iterate over all data blocks for all threads in a thread block, fetch one data block at a time in a way that different threads request consecutive but non-conflicting parts of the data block, and then, after all data blocks are fetched, let each thread work on its respective blocks independently. This is feasible since threads in a warp (or half-warp) execute the same stream of instructions (SIMT). Figure 10 depicts how threads in a half-warp cooperate with each other to fetch different blocks sequentially in time.

In order to ensure that the requests made by different threads when fetching different parts of the same data block do not conflict, we followed the best practices suggested by the device manufacturer to ensure these requests correspond to a single access to one row in a bank [6, 7, 42]. In particular, Shredder lets multiple threads of a half-warp read a contiguous memory interval simultaneously, under following conditions: (i) the size of the memory element accessed by each thread is either 4, 8, or 16 bytes; (ii) the elements form a contiguous block of memory; i.e, the Nth element is accessed by the Nth thread in the half-warp; and (iii) the address of the first element is aligned at a boundary of a multiple of 16 bytes.

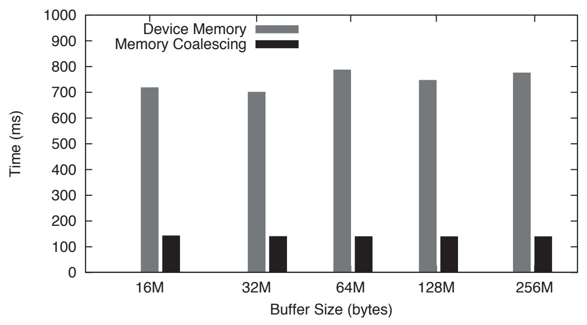


Figure 11: Normalized chunking kernel time with varied buffer-sizes for 1 GB data.

Effectiveness. Figure 11 shows the effectiveness of the memory coalescing optimization, where we compare the execution time for the chunking kernel using the normal device memory access and the optimized version. The results show that we improve performance by a factor of 8 by reducing bank conflicts. Since the granularity of memory coalescing is 48 KB (which is the size for the shared memory per thread block), we do not see any impact from varying buffer sizes (16 MB to 512 MB), and the benefits are consistent across different buffer sizes.

5 Implementation and Evaluation

We implemented Shredder in CUDA [6], and for an experimental comparison, we also implemented an optimized parallel host-only version of content-based chunking. This section describes these implementations and evaluates them.

5.1 Host-Only Chunking using `pthreads`

We implemented a library for parallel content-based chunking on SMPs using POSIX pthreads. We derived parallelism by creating pthreads that operate in different data regions using a Single Program Multiple Data (SPMD) strategy and communicate using a shared memory data structure. At a high level, the implementation works as follows: (1) divide the input data equally in fixed-size regions among N threads; (2) invoke the Rabin fingerprint-based chunking algorithm in parallel on N different regions; (3) synchronize neighboring threads in the end to merge the resulting chunk boundaries.

An issue that arises is that dynamic memory allocation can become a bottleneck due to the the serialization required to avoid race conditions. To address this, we used the Hoard memory allocator [12] instead of `malloc`.

5.2 Shredder Implementation

The Shredder library implementation comprises two main modules, the *host driver* and the *GPU kernel*. The host driver runs the control part of the system as a multi-threaded process on the host CPU running Linux. The GPU kernel uses one or more GPUs as co-processors for accelerating the SIMT code, and is implemented using the CUDA programming model from the NVidia GPGPU toolkit [6]. Next we explain key implementation details for both modules.

5.2.1 Host Driver

The host driver module is responsible for reading the input data either from the network or the disk and transferring the data to the GPU memory. Once the data is transferred then the host process dispatches the GPU kernel code in the form of RPCs supported by the CUDA toolkit. The host driver has two types of functionality: (1) the Reader/Store threads deal with reading and writing data from and to I/O channels; and (2) the Transfer thread is responsible for moving data between the host and the GPU memory. We implemented the Reader/Store threads using Asynchronous I/O and the Transfer thread using CUDA RPCs and page-pinned memory.

Asynchronous I/O (AIO). With asynchronous non-blocking I/O, it is possible to overlap processing and I/O by initiating multiple transfers at the same time. In AIO, the read request returns immediately, indicating that the read was successfully initiated. The application can then perform other processing while the background read operation completes. When the read response arrives, a signal registered with the read request is triggered to signal the completion of the I/O transaction.

Since the Reader/Store threads operate at the granularity of buffers, a single input file I/O may lead to issuing multiple `aio-read` system calls. To minimize the overhead of multiple context switches per buffer, we used `lio-listio` to initiate multiple transfers at the same time in the context of a single system call (meaning one kernel context switch).

5.2.2 GPU Kernel

The GPU kernel can be trivially derived from the C equivalent code by implementing a collection of functions in equivalent CUDA C with some assembly annotations, plus different access mechanisms for data layout in the GPU memory. However, an efficient implementation of the GPU kernel requires a bit more understanding of vector computations and the GPU architecture. We briefly describe some of these considerations.

Kernel optimizations. We have implemented minor kernel optimizations to exploit vector computation in GPUs. In particular, we used loop unrolling and instruction-level optimizations for the core Rabin fingerprint block. These changes are important because of the simplified GPU architecture, which lacks out-of-order execution, pipeline stalling in register usage, or instruction reordering to eliminate Read-after-Write (RAW) depen-

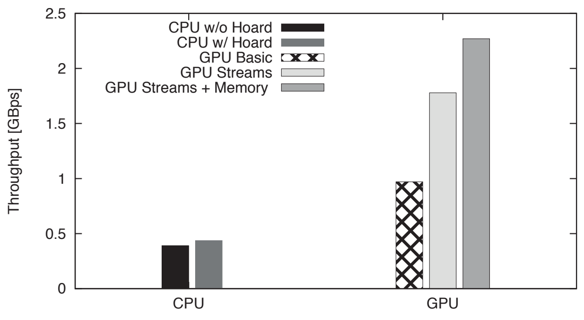


Figure 12: Throughput comparison of content-based chunking between CPU and GPU versions.

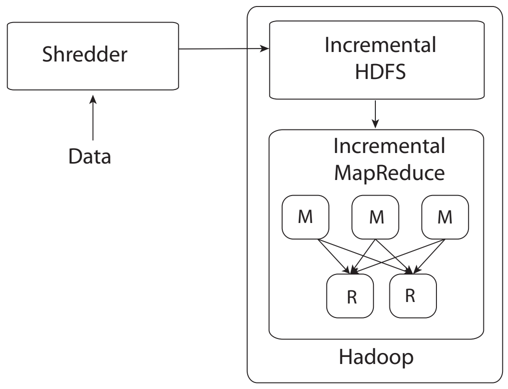


Figure 13: Incremental computations using Shredder.

dencies.

Warp divergence. Since the GPU architecture is Single Instruction Multiple Threads (SIMT), if threads in a warp diverge on a data-dependent conditional branch, then the warp is serially executed until all threads in it converge to the same execution path. To avoid a performance dip due to this divergence in warp execution, we carefully restructured the algorithm to have little code divergence within a warp, by minimizing the code path under data-dependent conditional branches.

5.3 Evaluation of Shredder

We now present our experimental evaluation of the performance of Shredder.

Experimental setup. We used a fermi-based GPU architecture, namely the Tesla C2050 GPU consisting of 448 processor cores (SPs). It is organized as a set of 14 SMs each consisting of 32 SPs running at 1.15 GHz. It has 2.6 GB of off-chip global GPU memory providing a peak memory bandwidth of 144 GB/s. Each SM has 32768 registers and 48 KB of local on-chip shared memory, shared between its scalar cores.

We also used an Intel Xeon processor based system as the host CPU machine. The host system consists of 12 Intel(R) Xeon(R) CPU X5650 @ 2.67 GHz with 48 GB of main memory. The host machine is running Linux with kernel 2.6.38 in 64-bit mode, additionally patched with GPU direct technology [4] (for SAN devices). The GCC 4.3.2 compiler (with -O3) was used to compile the source code of the host library. The GPU code is compiled using the CUDA toolkit 4.0 with NVidia driver version 270.41.03. The posix implementation is run with 12 threads.

Results. We measure the effectiveness of GPU-accelerated content-based chunking by comparing the performance of different versions of the host-only and GPU based implementation, as shown in Figure 12. We compare the chunking throughput for the pthreads implementation with and without using the Hoard memory allocator. For the GPU implementation, we compared the performance of the system with different optimizations turned on, to gauge their effectiveness. In particular, GPU Basic represents a basic implementation without any optimizations. The GPU Streams version includes the optimization to remove host and device bottlenecks using double buffering and a 4-stage pipeline. Lastly GPU Streams + Memory represents a version with all optimizations, including memory coalescing.

Our results show that a naive GPU implementation can lead to a $2X$ improvement over a host-only optimized implementation. The observation clearly highlights the potential of GPUs to alleviate computational bottlenecks. However, this implementation does not remove chunking as a bottleneck since SAN bandwidths on typical data servers exceed 10 Gbps. Incorporating the optimizations lead to Shredder outperforming the host-only implementation by a factor of over $5X$.

6 Case Study I: Incremental Computations

This section presents a case study of applying Shredder in the context of incremental computations. First we review Incoop, a system for bulk incremental processing, and then describe how we used Shredder to improve it.

6.1 Background: Incremental MapReduce

Incoop [14] is a generic MapReduce framework for incremental computations. Incoop leverages the fact that data sets that are processed by bulk data processing frameworks like MapReduce evolve slowly, and often the same computation needs to be performed repeatedly on this changing data (such as computing PageRank on every new web crawl) [21, 32, 34]. Incoop aims at processing this data incrementally, by avoiding recomputing parts of the computation that did not change, and transparently, by being backwards-compatible with the interface used by MapReduce frameworks.

To achieve these goals, Incoop employs a fine-grained result reuse mechanism, which captures a dependence

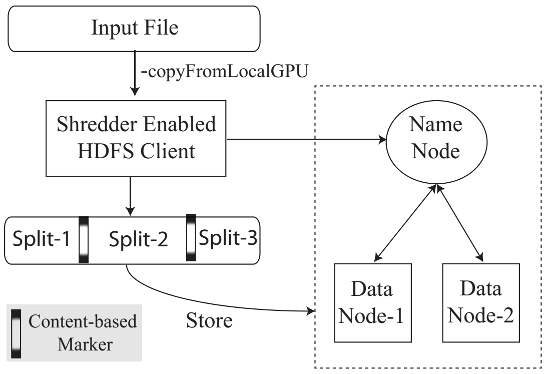


Figure 14: Shredder enabled chunking in Inc-HDFS.

graph among inputs and sub-computations, propagates changes along that graph so that only sub-computations that have changed need to be recomputed, and uses memoization to be able to reuse outputs from sub-computations whose inputs did not change. Incoop uses the Inc-HDFS file system (an extension to HDFS) to identify changes in the input and propagate them.

6.2 GPU-Accelerated Incremental HDFS

We use Shredder to support Incoop by designing a GPU-accelerated Incremental HDFS (Inc-HDFS), which is integrated with Incoop as shown in Figure 13. Inc-HDFS leverages Shredder to perform content-based chunking instead of using fixed-size chunking as in the original HDFS, thus ensuring that small changes to the input lead to small changes in the set of chunks that are provided as input to Map tasks. This enables the results of the computations performed by most Map tasks to be reused.

6.3 Implementation and Evaluation

We built our prototype GPU-accelerated Inc-HDFS on Hadoop-0.20.2. It is implemented as an extension to HDFS, where the computationally expensive chunking is offloaded to the Shredder-enabled HDFS client (as shown in Figure 14), before uploading chunks to the respective data nodes that will be storing them.

Inc-HDFS client. We integrated the Shredder library with Inc-HDFS client using a JAVA-CUDA interface. Once the data upload function is invoked, the Shredder library notifies the chunk boundaries to the Store thread, which in turn pushes the chunks from the memory of the client to the data nodes of HDFS.

Semantic chunking framework. The default behavior of the Shredder library is to split the input file into variable-length chunks based on the contents. However, since chunking is oblivious to the semantics of the input data, this could cause chunk boundaries to be placed anywhere, including, for instance, in the middle of a record that should not be broken. To address this, we leverage the fact that the MapReduce framework relies on the `InputFormat` class of the job to split up the input file(s) into logical InputSplits, each of which is then assigned to an individual Map task. We reuse this class to ensure that we respect the record boundaries in the chunking process.

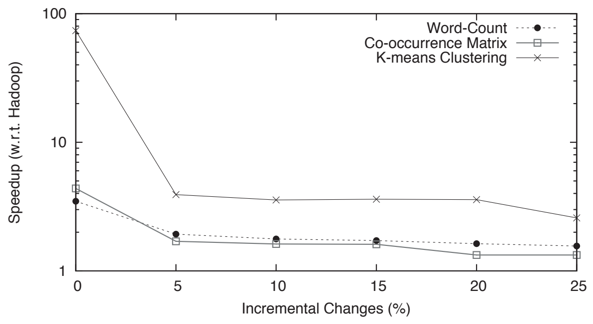


Figure 15: Speedup for incremental computation

HDFS shell. We extended the HDFS shell interface to invoke content-based chunking using the Shredder implementation. In particular, the shell interface offers new command (in addition to `copyFromLocal`) for uploading data in Inc-HDFS: `copyFromLocalGPU`.

Evaluation. We evaluated the effectiveness of incremental computations by measuring the speedups w.r.t. Hadoop for varying percentages of changes in the input data. Figure 15 shows the performance gains on a 20-node cluster, where all three MapReduce applications (K-means, Word-Count, Co-occurrence Matrix) show significant improvement in run-time for incremental runs. The effectiveness of the incremental approach degrades as the percentage of changes in the data set increases. Note that this experiment is not meant to highlight the speedup enabled by GPU acceleration, but instead shows how, after the data is chunked using Shredder, detecting duplicates at the storage level can imply savings in computation time.

7 Case Study II: Incremental Storage

In this section, we present our second case study where we use Shredder in the context of a consolidated incremental backup system.

7.1 Background: Cloud Backup

Figure 16 describes our target architecture, which is typical of cloud back-ends. Applications are deployed on virtual machines hosted on physical servers. The file system images of the virtual machines are hosted in a virtual machine image repository stored in a SAN volume. In this scenario, the backup process works in the following manner. Periodically, full image snapshots are taken for all the VM images that need to be backed up.

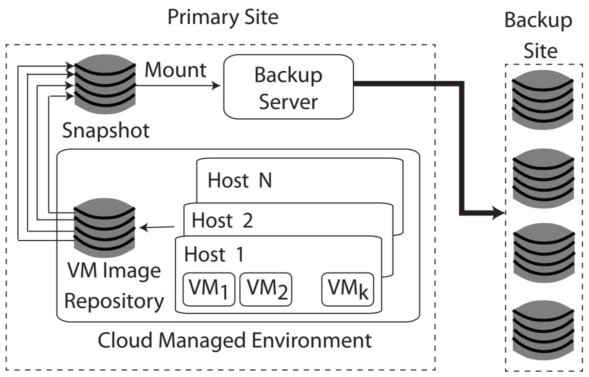


Figure 16: A typical cloud backup architecture

The core of the backup process is a backup server and a backup agent running inside the backup server. The image snapshots are mounted by the backup agent. The backup server performs the actual backup of the image snapshots onto disks or tapes. The consolidated or centralized data backup process ensures compliance of all virtual machines with the agreed upon backup policy. Backup servers typically have very high I/O bandwidth since, in enterprise environments, all operations are typically performed on a SAN [28]. Furthermore, the use of physical servers allows multiple dedicated ports to be employed solely for the backup process.

7.2 GPU-Accelerated Data Deduplication

The centralized backup process is eminently suitable for deduplication via content-based chunking, as most images in a data-center environment are standardized. Hence, virtual machines share a large number of files and a typical backup process would unnecessarily copy the same content multiple times. To exploit this fact, we integrate Shredder with the backup server, thus enabling data to be pushed to the backup site at a high rate while simultaneously exploiting opportunities for savings.

The Reader thread on the backup server reads the incoming data and pushes that into Shredder to form chunks. Once the chunks are formed, the Store thread computes a hash for the overall chunk, and pushes the chunks in the backup setup as a separate pipeline stage. Thereafter, these hashes collected for the chunks are batched together to enqueue in an index lookup queue. Finally, a lookup thread picks up the enqueued chunk fingerprints and looks up in the index whether a particular chunk needs to be backed up or is already present in the backup site. If a chunk already exists, a pointer to the original chunk is transferred instead of the chunk data. We deploy an additional Shredder agent residing on the backup site, which receives all the chunks and pointers and recreates the original uncompressed data. The overall architecture for integrating Shredder in a cloud

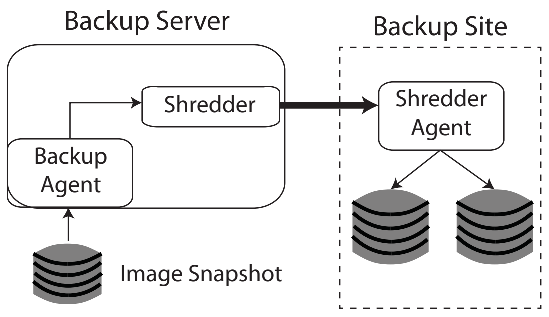


Figure 17: GPU-accelerated consolidated backup setup

backup system is described in Figure 17.

7.3 Implementation and Evaluation

Since high bandwidth fibre channel adapters are fairly expensive, we could not recreate the high I/O rate of modern backup servers in our testbed. Hence, we used a memory-driven emulation environment to experimentally validate the performance of Shredder. On our backup agent, we keep a master image in memory using memcached [5]. The backup agent creates new file system images from the master image by replacing part of the content from the master image using a predefined similarity table. The master image is divided into segments. The image similarity table contains a probability of each segment being replaced by a different content. The agent uses these probabilities to decide which segments in the master image will be replaced. The image generation rate is kept at 10 Gbps to closely simulate the I/O processing rate of modern X-series employed for I/O processing applications [28].

In this experiment, we also enable the requirement of a minimum and maximum chunk size, as used in practice by many commercial backup systems. As mentioned in Section 3, our current implementation of Shredder is not optimized for including a minimum and maximum chunk size, since the data that is skipped after a chunk boundary is still scanned for computing a Rabin fingerprint on the GPU, and only after all the chunk boundaries are collected will the Store thread discard all chunk boundaries within the minimum chunk size limit. As future work, we intend to address this limitation using more efficient techniques that were proposed in the literature [29, 31].

As a result of this limitation, we observe in Figure 18 that we are able to achieve a speedup of only $2.5X$ in backup bandwidth compared to the pthread implementation, but still we manage to keep the backup bandwidth close to the target 10 Gbps. The results also show that even though the chunking process operates independently of the degree of similarity in input data, the backup bandwidth decreases when the similarity between the data decreases. This is not a limitation

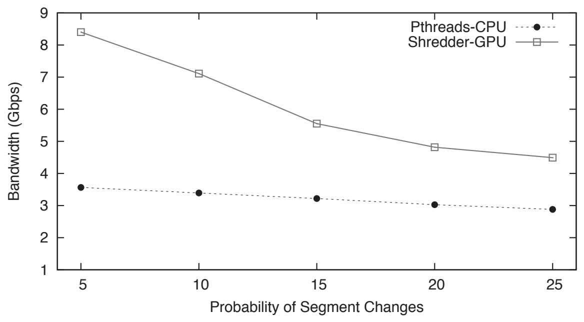


Figure 18: Backup bandwidth improvement due to Shredder with varying image similarity ratios

of our chunking scheme but of the unoptimized index lookup and network access, which reduces the backup bandwidth. Combined with optimized index maintenance (e.g., [17]), Shredder is likely to achieve the target backup bandwidth for the entire spectrum of content similarity.

8 Related Work

Our work builds on contributions from several different areas, which we briefly survey.

GPU-accelerated systems. GPUs were initially designed for graphics rendering, but, because of their cost-effectiveness, they were quickly adopted by the HPC community for scientific computations [3, 35]. Recently, the systems research community has leveraged GPUs for building other systems. In particular, PacketShader [23] is a software router for general packet processing, and SSLShader [26] uses GPUs in web servers to efficiently perform cryptographic operations. GPUs have also been used to accelerate functions such as pattern matching [44], network coding [43], and complex cryptographic operations [24]. In our work, we explored the potential of GPUs for large scale data, which raises challenges due to the overheads of data transfer. Recently, GPUs were used in software-based RAID controllers [16] for performing high-performance calculations of error correcting codes. However, this work does not propose optimizations for efficient data transfer.

Incremental Computations. Since modifying the output of a computation incrementally is asymptotically more efficient than recomputing everything from scratch, researchers and practitioners have built a wide range of systems and algorithms for incremental computations [14, 21, 25, 32, 34, 36, 37]. Our proposal speeds up the process of change identification in the input and is complementary to these systems.

Incremental Storage. Data deduplication is commonly used in storage systems. In particular, there is a large body of research on efficient index management [13, 17, 30, 46, 47]. In this paper, we focus on the complementary problem of content-based chunking [20, 27, 33]. High throughput content-based chunking is particularly relevant in environments that use SANs, where chunking can become a bottleneck. To overcome this bottleneck, systems have compromised the deduplication efficiency with sampling techniques or fixed-size chunking, or they have tried to scale chunking by deploying multi-node systems [15, 18, 19, 45]. A recent proposal shows that multi-node systems not only incur a high cost but also increase the reference management burden [22]. As a result, building a high throughput, cost-effective, single node systems becomes more important. Our system can be seen as an important step in this direction.

Network Redundancy Elimination. Content-based chunking has also been proposed in the context of redundancy elimination for content distribution networks (CDNs), to reduce the bandwidth consumption of ISPs [9, 10, 11, 38]. Also, many commercial vendors (such as Riverbed, Juniper, Cisco) offer middleboxes to improve bandwidth usage in multi-site enterprises, data centers and ISP links. Our proposal is complementary to this work, since it can be used to improve the throughput of redundancy elimination in such solutions.

9 Conclusions and Future Work

In this paper we have presented Shredder, a novel framework for content-based chunking using GPU acceleration. We applied Shredder to two incremental storage and computation applications, and our experimental results show the effectiveness of the novel optimizations that are included in the design of Shredder.

There are several interesting avenues for future work. First, we would like to incorporate into the library several optimizations for parallel content-based chunking [29, 31]. Second, our proposed techniques need to continuously adapt to changes in the technologies that are used by GPUs, such as the use of high-speed InfiniBand networking, which enables further optimizations in the packet I/O engine using GPU-direct [4]. Third, we would like explore new applications like middleboxes for bandwidth reduction using network redundancy elimination [10]. Finally, we would like to incorporate Shredder as an extension to recent proposals to devise new operating system abstractions to manage GPUs [41].

Acknowledgments

We thank Remzi Arpaci-Dusseau, the sysnets group members at MPI-SWS, Rose Hoberman, the anonymous reviewers, and our shepherd, Mark Lillibridge, for comments and suggestions on this paper. We thank Krishna Gummadi for granting access to the GPU hardware.

References

[1] The data deluge. http://www.economist.com/node/15579717, Feb. 2010.

[2] Calculating Memory System Power for DDR3. http://download.micron.com/pdf/technotes/ddr3/TN41_01DDR3 %20Power.pdf, Jan. 2012.

[3] General Purpose computation on GPUs. http://www.gpgpu.org, Jan. 2012.

[4] GPUDirect. http://developer.nvidia.com/gpudirect, Jan. 2012.

[5] Memcached. http://memcached.org/, Jan. 2012.

[6] NVidia CUDA. http://developer.nvidia.com/cuda-downloads, Jan. 2012.

[7] NVidia CUDA Tutorial. http://people.maths.ox.ac.uk/g̃ilesm/-hpc/NVIDIA/NVIDIA_CUDA_Tutorial_No_NDA_Apr08.pdf, Jan. 2012.

[8] Using the RDTSC Instruction for Performance Monitoring - Intel Developers Application Notes . http://www.ccsl.carleton.ca/ jamuir/rdtscpm1.pdf, Jan. 2012.

[9] AGGARWAL, B., AKELLA, A., ANAND, A., BALACHANDRAN, A., CHITNIS, P., MUTHUKRISHNAN, C., RAMJEE, R., AND VARGHESE, G. EndRE: an end-system redundancy elimination service for enterprises. In *Proceedings of the 7th USENIX conference on networked systems design and implementation* (Berkeley, CA, USA, 2010), NSDI'10, USENIX Association, pp. 28–28.

[10] ANAND, A., GUPTA, A., AKELLA, A., SESHAN, S., AND SHENKER, S. Packet caches on routers: the implications of universal redundant traffic elimination. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication* (New York, NY, USA, 2008), SIGCOMM '08, ACM, pp. 219–230.

[11] ANAND, A., SEKAR, V., AND AKELLA, A. Smartre: an architecture for coordinated network-wide redundancy elimination. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication* (New York, NY, USA, 2009), SIGCOMM '09, ACM, pp. 87–98.

[12] BERGER, E. D., MCKINLEY, K. S., BLUMOFE, R. D., AND WILSON, P. R. Hoard: a scalable memory allocator for multithreaded applications. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2000), ASPLOS'00, ACM, pp. 117–128.

[13] BHAGWAT, D., ESHGHI, K., LONG, D. D. E., AND LILLIBRIDGE, M. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Proceedings of the 17th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS'09* (2009).

[14] BHATOTIA, P., WIEDER, A., RODRIGUES, R., ACAR, U. A., AND PASQUIN, R. Incoop: Mapreduce for incremental computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (New York, NY, USA, 2011), SOCC '11, ACM, pp. 7:1–7:14.

[15] CLEMENTS, A. T., AHMAD, I., VILAYANNUR, M., AND LI, J. Decentralized deduplication in san cluster file systems. In *Proceedings of the 2009 conference on USENIX Annual technical conference* (Berkeley, CA, USA, 2009), USENIX'09, USENIX Association, pp. 8–8.

[16] CURRY, M. L., WARD, H. L., SKJELLUM, A., AND BRIGHTWELL, R. A lightweight, gpu-based software raid system. In *Proceedings of the 2010 39th International Conference on Parallel Processing* (Washington, DC, USA, 2010), ICPP '10, IEEE Computer Society, pp. 565–572.

[17] DEBNATH, B., SENGUPTA, S., AND LI, J. Chunkstash: speeding up inline storage deduplication using flash memory. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference* (Berkeley, CA, USA, 2010), USENIX'10, USENIX Association, pp. 16–16.

[18] DONG, W., DOUGLIS, F., LI, K., PATTERSON, H., REDDY, S., AND SHILANE, P. Tradeoffs in scalable data routing for deduplication clusters. In *Proceedings of the 9th USENIX conference on File and stroage technologies* (Berkeley, CA, USA, 2011), FAST'11, USENIX Association, pp. 2–2.

[19] DUBNICKI, C., GRYZ, L., HELDT, L., KACZMARCZYK, M., KILIAN, W., STRZELCZAK, P., SZCZEPKOWSKI, J., UNGUREANU, C., AND WELNICKI, M. Hydrastor: a scalable secondary storage. In *Proccedings of the 7th conference on File and storage technologies* (Berkeley, CA, USA, 2009), FAST'09, USENIX Association, pp. 197–210.

[20] ESHGHI, K., AND TANG, H. K. A Framework for Analyzing and Improving Content-Based Chunking Algorithms. Tech. Rep. HPL-2005-30R1, HP Technical Report, 2005.

[21] GUNDA, P. K., RAVINDRANATH, L., THEKKATH, C. A., YU, Y., AND ZHUANG, L. Nectar: automatic management of data and computation in datacenters. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 1–8.

[22] GUO, F., AND EFSTATHOPOULOS, P. Building a high-performance deduplication system. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference* (Berkeley, CA, USA, 2011), USENIX'11, USENIX Association, pp. 25–25.

[23] HAN, S., JANG, K., PARK, K., AND MOON, S. Packetshader: a gpu-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 conference on SIGCOMM* (New York, NY, USA, 2010), SIGCOMM '10, ACM, pp. 195–206.

[24] HARRISON, O., AND WALDRON, J. Practical symmetric key cryptography on modern graphics hardware. In *Proceedings of the 17th conference on Security symposium* (Berkeley, CA, USA, 2008), USENIX Security'08, USENIX Association, pp. 195–209.

[25] HE, B., YANG, M., GUO, Z., CHEN, R., SU, B., LIN, W., AND ZHOU, L. Comet: batched stream processing for data intensive distributed computing. In *Proceedings of the 1st ACM symposium on Cloud computing* (New York, NY, USA, 2010), SoCC '10, ACM, pp. 63–74.

[26] JANG, K., HAN, S., HAN, S., MOON, S., AND PARK, K. Sslshader: cheap ssl acceleration with commodity processors. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation* (Berkeley, CA, USA, 2011), NSDI'11, USENIX Association, pp. 1–1.

[27] KRUUS, E., UNGUREANU, C., AND DUBNICKI, C. Bimodal content defined chunking for backup streams. In *Proceedings of the 8th USENIX conference on File and storage technologies* (Berkeley, CA, USA, 2010), FAST'10, USENIX Association, pp. 18–18.

[28] KULKARNI, V. Delivering on the i/o bandwidth promise: over 10gb/s large sequential bandwidth on ibm x3850/x3950 x5. Tech. rep., IBM, 2010.

[29] LILLIBRIDGE, M. Parallel processing of input data to locate landmarks for chunks, 16 August 2011. U.S. Patent No. 8,001,273.

[30] LILLIBRIDGE, M., ESHGHI, K., BHAGWAT, D., DEOLALIKAR, V., TREZISE, G., AND CAMBLE, P. Sparse indexing: large scale, inline deduplication using sampling and locality. In *Proccedings of the 7th conference on File and storage technologies* (Berkeley, CA, USA, 2009), FAST'09, USENIX Association, pp. 111–123.

[31] LILLIBRIDGE, M., ESHGHI, K., AND PERRY, G. Producing chunks from input data using a plurality of processing elements, 12 July 2011. U.S. Patent No. 7,979,491.

[32] LOGOTHETIS, D., OLSTON, C., REED, B., WEBB, K. C., AND YOCUM, K. Stateful bulk processing for incremental analytics. In *Proceedings of the 1st ACM symposium on Cloud computing* (New York, NY, USA, 2010), SoCC '10, ACM, pp. 51–62.

[33] MUTHITACHAROEN, A., CHEN, B., AND MAZIÈRES, D. A low-bandwidth network file system. In *Proceedings of the eighteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2001), SOSP '01, ACM, pp. 174–187.

[34] OLSTON, C., CHIOU, G., CHITNIS, L., LIU, F., HAN, Y., LARSSON, M., NEUMANN, A., RAO, V. B., SANKARASUBRAMANIAN, V., SETH, S., TIAN, C., ZICORNELL, T., AND WANG, X. Nova: continuous pig/hadoop workflows. In *Proceedings of the 2011 international conference on Management of data* (New York, NY, USA, 2011), SIGMOD '11, ACM, pp. 1081–1090.

[35] OWENS, J. D., LUEBKE, D., GOVINDARAJU, N., HARRIS, M., KRGER, J., LEFOHN, A., AND PURCELL, T. J. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum 26*, 1 (2007), 80–113.

[36] PENG, D., AND DABEK, F. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 1–15.

[37] POPA, L., BUDIU, M., YU, Y., AND ISARD, M. DryadInc: Reusing work in large-scale computations. In *1st USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '09)* (June 15 2009).

[38] PUCHA, H., ANDERSEN, D. G., AND KAMINSKY, M. Exploiting similarity for multi-source downloads using file handprints. In *Proceedings of the 4th USENIX conference on Networked systems design and implementation* (Berkeley, CA, USA, 2007), NSDI'07, USENIX Association, pp. 2–2.

[39] QUINLAN, S., AND DORWARD, S. Awarded best paper! - venti: A new approach to archival data storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2002), FAST '02, USENIX Association.

[40] RABIN, M. O. Fingerprinting by random polynomials. Tech. rep., 1981.

[41] ROSSBACH, C. J., CURREY, J., SILBERSTEIN, M., RAY, B., AND WITCHEL, E. Ptask: operating system abstractions to manage gpus as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 233–248.

[42] SAXENA, V., SABHARWAL, Y., AND BHATOTIA, P. Performance evaluation and optimization of random memory access on multicores with high productivity. In *International Conference on High Performance Computing (HiPC)* (2010), IEEE.

[43] SHOJANIA, H., LI, B., AND WANG, X. Nuclei: Gpu-accelerated many-core network coding. In *Proc. of IEEE Infocom , Rio de Janeiro* (2009), INFOCOM'09, pp. 459–467.

[44] SMITH, R., GOYAL, N., ORMONT, J., SANKARALINGAM, K., AND ESTAN, C. Evaluating gpus for network packet signature matching. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software, ISPASS'09* (2009).

[45] UNGUREANU, C., ATKIN, B., ARANYA, A., GOKHALE, S., RAGO, S., CALKOWSKI, G., DUBNICKI, C., AND BOHRA, A. Hydrafs: a high-throughput file system for the hydrastor content-addressable storage system. In *Proceedings of the 8th USENIX conference on File and storage technologies* (Berkeley, CA, USA, 2010), FAST'10, USENIX Association, pp. 17–17.

[46] 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 *Proceedings of the 2011 USENIX conference on USENIX annual technical conference* (Berkeley, CA, USA, 2011), USENIX'11, USENIX Association, pp. 26–28.

[47] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2008), FAST'08, USENIX Association, pp. 18:1–18:14.

Adding Advanced Storage Controller Functionality via Low-Overhead Virtualization

Muli Ben-Yehuda, Michael Factor, Eran Rom, and Avishay Traeger
IBM Research–Haifa
{muli,factor,eranr,avishay}@il.ibm.com

Eran Borovik and Ben-Ami Yassour
{eran.borovik,benami.yassour}@gmail.com

Abstract

Historically, storage controllers have been extended by integrating new code, e.g., file serving, database processing, deduplication, etc., into an existing base. This integration leads to complexity, co-dependency and instability of both the original and new functions. Hypervisors are a known mechanism to isolate different functions. However, to enable extending a storage controller by providing new functions in a virtual machine (VM), the virtualization overhead must be negligible, which is not the case in a straightforward implementation. This paper demonstrates a set of mechanisms and techniques that achieve near zero runtime performance overhead for using virtualization in the context of a storage system.

1 Introduction

Additional functions, such as file serving or database, are often added to existing storage systems to meet new requirements. Historically, this has been done via code integration, or by running the new function on a gateway or virtual storage appliance (VSA [37]). Code integration generally performs best. However, the new function must run on the same OS version, the controller's main functionality is vulnerable to bugs due to lack of isolation, resource management is complicated for software which assumes a dedicated system, and development complexity increases in particular when the new function already exists as independent software. The gateway approach offers isolation but adds both latency and hardware costs.

A hypervisor can isolate the new function, allow for differing OS versions, and simplify development. However, until now the high performance overhead of virtualization (in particular virtualized I/O) has made this approach impractical. In this paper, we show how to use server-based virtualization to integrate new functions into a storage system with near zero performance cost. Our approach is in line with the VSA approach, but we run the VM directly on the storage system.

While our work was done using KVM [14], our insights are not KVM-specific. We do take advantage of the fact that KVM uses an asymmetric model in which some of the code is virtualized (the new features) while other code (the original storage system) runs on "bare metal," unaware of the existence of the hypervisor.

There are three sources of performance overhead. *Base overheads* include aspects such as virtual memory management or process switching. *External communication* with storage clients is important when the new function is a "filter" on top of the original storage system, e.g., a file server. Finally, *internal communication* overheads are incurred to tie the new function to the original controller.

To reduce base overhead, we use two main techniques. First, we statically allocate CPU cores to the guest to ensure that the function has sufficient resources. Second, we statically allocate memory for the VM, backing that area with larger pages to reduce translation overheads.

The straightforward implementation of external communication is expensive because the hypervisor intervenes when physical events occur (e.g., interrupts or device accesses). Each such intervention entails an expensive "exit" from the guest code to the hypervisor. The highest-performing approach for reducing this overhead is device assignment, which eliminates exits for device access. Thus, to reduce these costs, we assign the network device directly to the guest using an SR-IOV-enabled adapter [23] which allows the guest to send requests directly to the device. To eliminate exits for interrupts, we use polling instead of interrupts, a well-known technique in storage systems.

To reduce the cost of internal communication, we modified KVM's para-virtual block driver to poll as well, eliminating exits due to PIOs and interrupt injections. This provides for a fast, exit-less, zero-copy transport.

By using these techniques, we show no measurable difference in network latency between bare metal and virtualized I/O and under 5% difference in throughput. For internal communication, micro-benchmarks show 6.6μs latency overhead, read throughput of 357K IOPS, and write throughput of 284K IOPS; roughly seven times better than a base KVM implementation. In addition, an I/O intensive filer workload running in KVM incurs less than 0.4% runtime performance overhead compared to bare metal integration.

Our main contributions are:

- a detailed, benchmark-driven analysis of virtualization overheads in a storage system context,
- a set of approaches to removing overheads, and
- a demonstration of how these approaches enable running new storage features in a VM with essentially zero runtime performance overhead.

The rest of the paper is organized as follows. Section 2 provides background on KVM and VM I/O. We take an incremental approach to show our performance improvements; Section 3 describes the experimental environment. Sections 4 and 5 present a performance analysis and describe optimizations related to the external and internal communication interfaces, respectively. Base overheads are shown together with macro-benchmark results are in Section 6. Section 7 describes related work and we conclude in Section 8.

2 x86 I/O Virtualization Primer

We now provide some background information on KVM (the hypervisor used in this paper) and virtual machine I/O. There are two main options for where a hypervisor resides. Type 1 hypervisors run directly on the hardware, whereas type 2 hypervisors are hosted by an OS. KVM takes a hybrid approach that combines the benefits of both. It is a Linux kernel module that leverages Intel VT-x or AMD-V CPU features for running unmodified virtual machines, thereby creating a single host kernel/hypervisor that runs both processes and virtual machines. Such a hybrid architecture allows the storage controller software to run unmodified on bare metal while also running additional functionality in virtual machines.

There are three main methods for accessing I/O devices in VMs. In the first, *emulation*, the hypervisor emulates a specific device in software [35]. The OS running in the VM (guest OS) uses its regular device drivers to access the emulated device. This method requires no changes to the guest, but suffers from poor performance.

In the second method, *para-virtualization* [4], the guest OS runs specialized code to cooperate with the hypervisor to reduce overheads. For example, KVM's para-virtualized drivers use *virtio* [26], which presents a ring buffer transport (*vring*) and device configuration as a PCI device. Drivers such as network, block, and video are implemented using virtio. In general, the guest OS driver places pointers to buffers on the vring and initiates I/O via a Programmed I/O (PIO) command. The hypervisor directly accesses the buffers from the guest OS's memory (zero-copy). Para-virtualized devices perform better than emulated devices, but require installing hypervisor-specific drivers in the guest OS.

The third method, *device assignment* [6, 17, 39], gives the VM a physical device that it can submit I/Os to without the hypervisor's involvement. An I/O Memory Management Unit (IOMMU) provides address translation and memory protection [6, 7, 38]. Interrupts, however, are routed to the guest OS via the hypervisor. Assigning a device to the VM means that no other OS can access it (including the hypervisor or other guests). However, technologies such as Single Root I/O Virtualization (SR-IOV) [23] allow devices to be assigned to multiple OSs.

3 Experimental Setup

We take an incremental approach to showing how to eliminate the virtualization overheads. For our experiments we used two servers, each with two quad-core 2.93GHz EPT-enabled Intel Xeon 5500 processors, 16GB of RAM and an Emulex OneConnect 10Gb Ethernet adapter. The servers were connected with a 10Gb cable. One server acted as a load generator and the other was our (emulated) storage controller platform.

We used RHEL 5.4 with the RedHat 2.6.18-164.el5 kernel for both the load server and the guest. The controller server used the RedHat kernel for bare metal runs and Ubuntu 9.10 with a vanilla 2.6.33 kernel for KVM runs. The newer kernel was necessary for running KVM.

The controller server was run with four cores enabled, unless otherwise specified. For VM-based experiments, two cores and 2GB of memory were assigned to the guest; all four cores were used by the host in the bare metal cases.

We used an 8GB ramdisk for the storage back-end in the experiments described in Section 5 and 6. This allowed us to measure I/O performance without physical disks becoming the bottleneck. We accessed the ramdisk via a loopback device, which allowed us to assign disk I/O handling to specific cores, similar to the way a storage controller functions.

All results shown are the averages of at least 5 runs, with standard deviations below 5%.

4 Network Communication Performance

Enabling the guest to interact with the outside world requires I/O access. As discussed in Section 2, each of the three common approaches to I/O virtualization has benefits and drawbacks. We identified device assignment—the best performing option—as the most suitable approach for adding new functionality to storage controllers. KVM's initial device assignment implementation, however, did not provide the necessary performance. In the remainder of this section, we analyze device assignment and discuss a set of optimizations which allowed us to achieve near bare-metal performance.

Virtualization overhead is mainly due to events that are trapped by the hypervisor, causing costly *exits* [1, 5, 16]. The overhead is a factor of the frequency of exits and the time it takes the hypervisor to handle the exit and resume running the guest. To examine the performance impact of virtualization for our intended use and ways to reduce it, we focused on networking micro-benchmarks. Our goal is to minimize the amount of time that the hypervisor needs to run, by minimizing the number of exits.

The first technique that we used to improve the guest's performance is related to the handling of the `hlt` (halt) and `mwait` x86 instructions. When the OS does not have any work to do it can call these instructions to enter a power saving mode. Most hypervisors will trap these commands and will run other tasks on the core. In our case, however, the new function should always run. We therefore instructed the guest OS to enter an idle loop when there is no work to be done by enabling a kernel boot parameter (`idle=poll`). This improves performance, as the guest is always running.

The second technique that we used is related to interrupt handling. Most of the guest exits related to device assignment are caused by interrupts [2, 5, 18]. Every external interrupt causes at least two guest exits: first, when the interrupt arrives (causing the hypervisor to gain control and to inject the interrupt to the guest) and when the guest signals completion of the interrupt handling (causing the host to gain control and to emulate the completion for the guest). The guest can configure the adapter to use two different interrupt-delivery modes: MSI, which is the newer message based interrupt protocol, or the legacy INTX protocol. The KVM implementation we used incurred additional overhead when using MSI interrupts, due to additional exits for masking and unmasking adapter interrupts. Since most of the virtualization overhead comes from interrupts, our approach is to run the adapter in polling rather than interrupt-driven mode.

In Linux today, most network adapters use NAPI [30, 31], a hybrid approach to reducing interrupt overhead which switches between polling and interrupt-driven operation depending on the network traffic. However, even with NAPI, we have seen interrupt rates of 70K interrupts per second. Since such a high interrupt rate can incur prohibitive overhead and interrupts are not necessary for our intended use case, we decided to forgo interrupts and use polling. Our polling driver creates a new thread for the polling functions. The adapter we use has three types of events: packet received, packet sent, and command completion. Since there is no way to know when a packet will be received, our polling driver continuously polls for packets received; packet sent and command completion indications are handled by the same polling thread every so often. Using a constantly polling thread means that we dedicate most of a core for this functionality. While this might seem expensive from the resources perspective, it proved critical to achieve the desired performance. A single core could also be used to poll multiple devices by integrating their polling threads into a single thread, or by scheduling different polling threads on the same core. We did not experiment with this configuration.

Next we evaluate the performance of the polling driver using network micro-benchmarks. Table 1 depicts the average duration time of a ping flood command going from a client machine to the system under test. The system under test replies to pings using our polling driver either in *polling* mode or in *INTX* mode. The driver runs either in the host (*bare-metal*), or in the *guest* with halt disabled, or in the guest with halt enabled (*guest halt*).

	Bare-metal	Guest	Guest halt
INTX	24	49	89
Polling	21	21	21

Table 1: Ping average latency (μs)

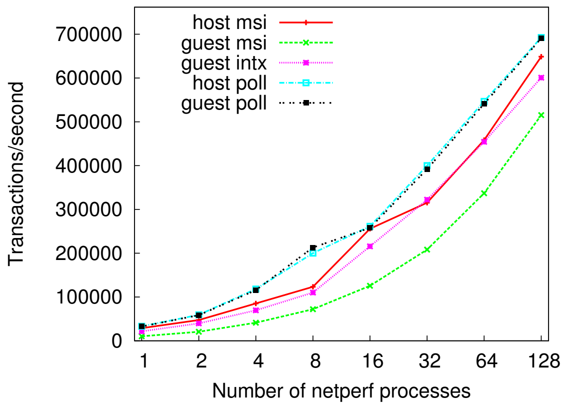


Figure 1: `netperf` request-response throughput

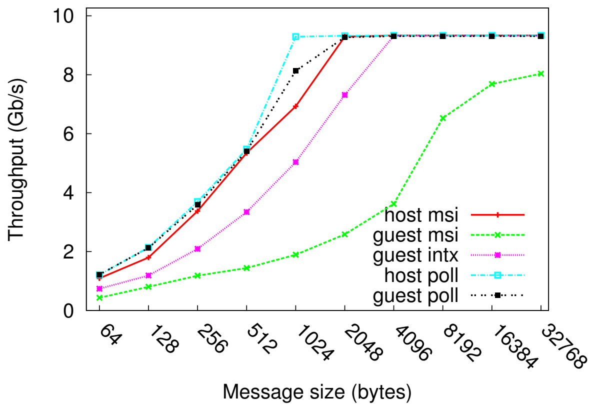


Figure 2: `netperf` TCP send throughput

Figure 1 shows the results for several `netperf` request-response configurations, measuring round-trip time using 1 byte packets. *guest msi* and *guest intx* stand for the guest using MSI and INTX interrupt delivery, respectively. *host msi* stands for the host using interrupts in MSI mode; *guest poll* and *host poll* stand for the guest and host using polling mode, respectively. As expected, polling mode achieves better performance than interrupt mode in the host (i.e, on bare metal). Since in guest mode the cost of interrupts is much higher, the gain from using polling is more significant than in the bare-metal case. Using MSI interrupts in guest mode has significant impact on the performance with this KVM version since there are frequent exits due to interrupt masking calls by the guest.

Figure 2 shows the results of a single-threaded `netperf` send TCP throughput test (system under test is sending) in the same configurations as the previous figure: host using polling and INTX interrupts, guest using

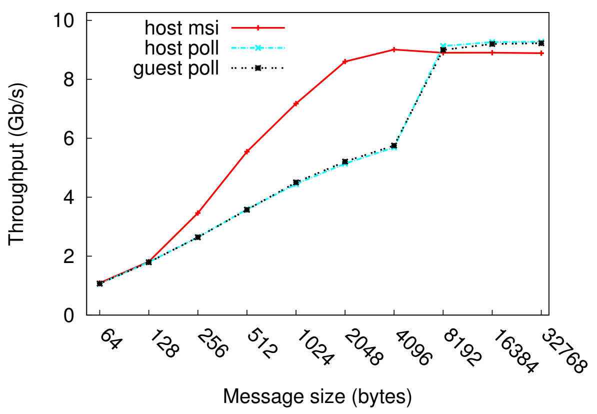


Figure 3: `netperf` *TCP receive throughput*

polling, INTX, and MSI interrupts. Here the contribution of polling is less noticeable, since the TCP stack batches network processing. On bare metal, polling provides better performance than interrupt mode. In guest mode, the advantage of polling is much more significant.

Figure 3 shows the results of a single-threaded `netperf` receive TCP throughput test (system under test is receiving). Here it is surprising to see that for the bare-metal case the performance of polling (*host poll*) is less than that of interrupts (*host msi*). The reason is that in the netperf throughput test the sender is the bottleneck. When the receiver is working in polling mode, it sends many more acknowledgment packets to the sender. For example in the case of 1K messages, the receiver sends approximately 10 times as many ACKs. Since the sender is already the bottleneck, sending more ACKs generates more load on the sender, which reduces sender throughput. While this issue is noticeable for this micro-benchmark, in practice, the handling time of a packet by the receiver is much larger, hence in most cases the sender is not the bottleneck. Polling achieves the same performance in the *guest poll* and *host poll* cases, which indicates that the virtualization-induced runtime overhead is negligible.

To verify that the reduced polling performance for the receive test is an artifact of TCP, we ran the same test using UDP. With UDP, all setups—guest or bare metal, interrupts or polling—achieve the same performance. Because the sender is the bottleneck, once the TCP ACK effect is removed, performance is not affected by the receiver's mode of operation.

5 Internal Communication Performance

Of the three methods for accessing I/O devices described in Section 2, we use para-virtualization for internal communication. Para-virtualization performs better than emulated devices, and because we supply the VM image that runs in the controller, we can easily use custom drivers. Further, our goal is to transmit I/O requests to a controller process running on the host, so device assignment is less

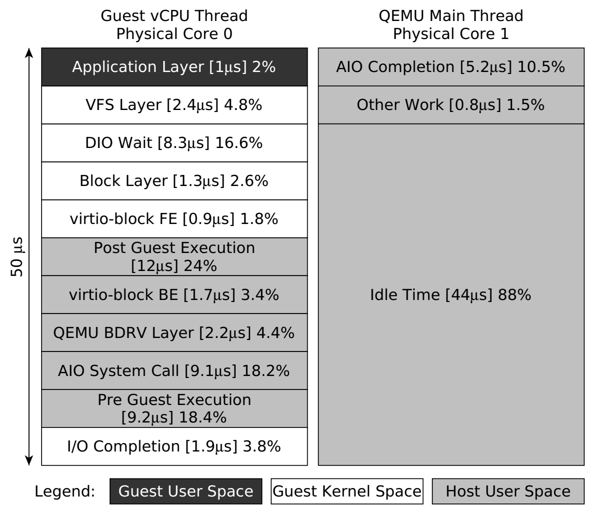

Guest vCPU Thread
Physical Core 0
QEMU Main Thread
Physical Core 1
50 µs
Application Layer [1µs] 2%
VFS Layer [2.4µs] 4.8%
DIO Wait [8.3µs] 16.6%
Block Layer [1.3µs] 2.6%
virtio-block FE [0.9µs] 1.8%
Post Guest Execution [12µs] 24%
virtio-block BE [1.7µs] 3.4%
QEMU BDRV Layer [2.2µs] 4.4%
AIO System Call [9.1µs] 18.2%
Pre Guest Execution [9.2µs] 18.4%
I/O Completion [1.9µs] 3.8%
AIO Completion [5.2µs] 10.5%
Other Work [0.8µs] 1.5%
Idle Time [44µs] 88%
Legend:
Guest User Space
Guest Kernel Space
Host User Space

Figure 4: Unmodified KVM para-virtualized block I/O path.

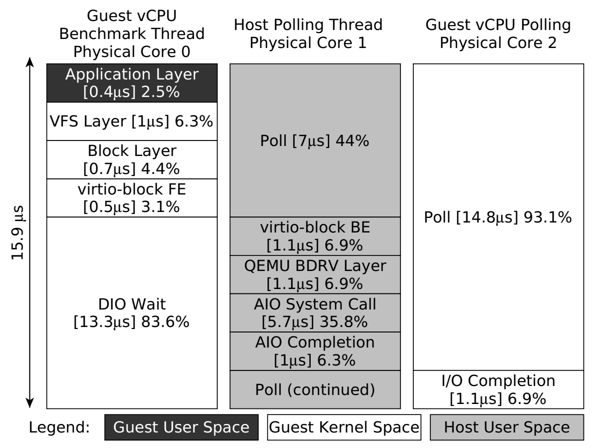


Figure 5: Para-virtualized block I/O path with polling.

practical. For example, we cannot assign the drives because the storage controller must "own" them, and not the guest OS. One may also consider using external communication to access the controller via iSCSI or Fibre Channel, but this adds unnecessary communication overheads.

We use ramdisk as the backing store for our analysis to prevent the disks from dominating latencies or becoming a bottleneck. In addition, we use direct I/O to prevent caching effects that mask virtualization overheads. Latencies presented are the average over 10 minutes.

Section 5.1 describes the vanilla KVM para-virtualized block I/O, and Section 5.2 describes our optimizations.

5.1 KVM Para-virtualized Block I/O

Figure 4 depicts the unmodified para-virtualized block I/O path in KVM, along with associated latencies for major code blocks when executing a 4KB direct I/O write request. The guest application initiates an I/O, which is handled by the guest kernel as usual. The direct I/O waiting time (DIO Wait, 16.6% of the total), consists of world switches and context switches between threads inside the guest. Though we have drawn it as one block, it is interleaved with other code running on the same core. The para-virtualized block driver (virtio-block front-end) iterates over the requests in the elevator queue and places each request's I/O descriptors on the *vring*, a queue residing in guest memory that is accessible by the host. The driver then issues a programmed I/O (PIO) command which causes a world switch from guest to host.

Control is transferred to the KVM kernel module to handle the exit. The post- and pre-execution times (24% and 18.4%, respectively) account for the work done by both KVM and QEMU to change contexts between the guest and QEMU process (including the exit/entry). KVM identifies the cause of the exit and, in this case, passes control to the QEMU virtio-block back-end (BE). It extracts the I/O descriptors from the vring without copies and passes the requests to the block driver layer (QEMU BDRV), which initiates asynchronous I/Os to the block device. The guest may now resume execution.

An event-driven dedicated QEMU thread receives I/O completions and forwards them to the virtio-block BE. The BE updates the vring with completion information and calls upon KVM to inject an interrupt into the guest, for which KVM must initiate a world switch. When the guest resumes, its kernel handles the interrupt as normal, and then accesses its APIC to signal the end of interrupt, causing yet another exit. Locks to synchronize the two QEMU threads incur additional overhead.

5.2 Para-virtualized Block Optimizations

To reduce virtualization overhead, we added a polling thread to QEMU as depicted in Figure 5. The thread polls the vring (1) for I/O requests coming from the guest and (2) for I/O completions coming from the host kernel. The polling thread invokes the virtio-block BE code on incoming I/Os and completions. This thread does not necessarily need to reside in QEMU; if the storage controller is polling-based, its polling thread may be used.

As discussed in Section 4, we added a thread to the guest which polls the networking device. We utilize this same thread to poll the vring for I/O completions. When it detects an I/O completion event, it invokes the guest I/O completion stack, which would normally be called by the interrupt handler. By using polling on both sides of the vring, we avoid all I/O-related exits, and thus also avoid all of the pre- and post-guest execution code. We also avoid locking the queue, since now only the polling thread accesses it. For the 4KB direct I/O write, this improves the latency from 50μs to 15.9μs.

Comparing Figures 4 and 5, we see that polling better utilizes the CPU for I/O-related work. Additionally, components that we didn't directly optimize (such as the VFS layer, for example) are more efficient thanks to better cache utilization and less cache pollution due to fewer context switches.

We performed two additional code optimizations in QEMU to reduce latencies, whose impact is already included in the above discussion. When accessing a guest's memory, QEMU must first translate the address using a page-table–like data structure. This handles cases where the guest's memory can be remapped (for example, when dealing with PCI devices). In our case, the memory layout is static, rendering the translation unnecessary. Removing unnecessary lookups improved performance by 4.6% for 4KB reads and 4.2% for 4KB writes. The second optimization is to use a memory pool for internal QEMU request structures. This saved 3% for 4KB reads and 2.5% for 4KB writes.

5.3 Overall Performance Calculation

A storage controller running a new function in a VM that uses interrupts for its internal communication would have a rather significant performance penalty. Looking at Figure 4, the corresponding storage controller implementation would look similar, except that the AIO calls would be replaced by asynchronous calls to the controller code. We consider any work done from the time the application submits the I/O until it reaches the controller to be virtualization overhead (work that would not be done if running directly on the host). In the unmodified case, the overhead is 49μs (we subtract only the latency of the application layer from the total).

If our techniques were integrated into a controller, we would calculate the latency overhead as follows, based on Figure 5. We begin with the total, 15.9μs, and subtract the application layer, as we did in the previous case. Further, we subtract the QEMU BDRV layer, and the AIO system call and completion, because these would be replaced by the controller code, and are therefore not considered virtualization overhead. The final overhead is therefore 7.7μs before the two QEMU optimizations, and 6.6μs after.

To put the overheads in context, we estimate our performance impact on the fastest latencies published using the SPC-1 benchmark since 2009 [34]. The fastest result was 130μs, and our virtualization technique would add approximately 5% overhead to this case (the baseline case would add approximately 38%). The average of the 27 controllers' fastest latencies is 482μs, and in this average case, our virtualization techniques would add only 1.4% (the baseline would add over 10%).

Our improvements affect throughput in addition to latency. To measure these effects, we ran microbenchmarks consisting of multi-threaded 4KB direct I/Os. For multi-threaded 4KB direct I/Os, we improved read IOPs by a factor of 7.3x (from 48.8K to 357.5K), and write IOPS by 6.5x (from 43.8K to 284.1K).

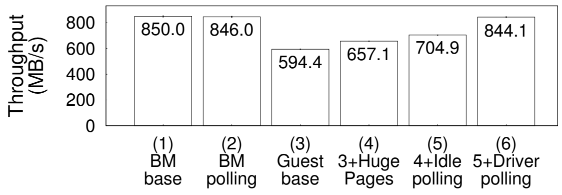


Figure 6: File server workload with 6 cores

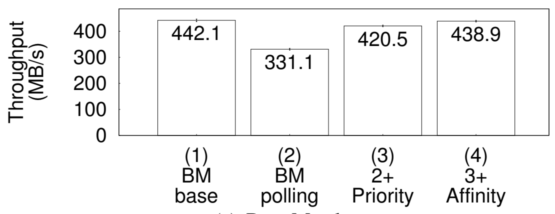
6 File Server Workload

We next tested the end-to-end performance of running a server in a VM on a storage controller. We ran a file server in our VM, and used dbench [36] v4.00 to generate 4KB NFS read requests which all arrived at the 10Gb NIC, went through the local file system and block layers, through the para-virtualized block interface, and were satisfied by the ramdisk on the host side. We always allocated two cores to the controller function, and either two or four to the file server (as specified). In the virtualized cases, all file server cores were given to the VM. All cores were fully utilized for all cases.

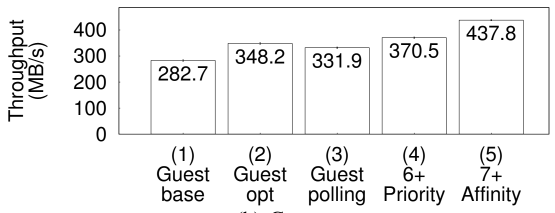
Figure 6 shows the results when running with six cores: two for the controller function and four for the file server. Bars 1 and 2 show the bare metal case without polling and with, respectively. Roughly the same performance is attained in both cases. The third bar shows the baseline measurement for the guest, which is a significant degradation as compared to the bare metal cases. We identified three main causes for this performance drop.

First, we noticed a large number of page faults on the host caused by the running VM. We mitigated this using the Linux kernel's *HugePages* mechanism, which backs a given process with 2MB pages instead of 4KB pages. This allows the OS to store fewer TLB page entries, resulting in fewer TLB faults and fewer EPT table lookups. HugePages improved performance by 10.5%, as shown in the fourth bar of Figure 6. A feature in a recent Linux kernel release makes the use of HugePages automatic [10]. The second issue affecting performance was halt exits, described in Section 4. We avoid these exits by setting the guest scheduler to poll. This further improved performance by 7.3% (fifth bar in Figure 6). The final performance improvement was to add driver polling, for both the network and block interfaces (described in Sections 4 and 5.2). This further improved performance by 19.7%, and brings the guest's performance to be statistically indistinguishable from bare metal.

Next, we ran the same workload, but this time allocated only two cores to the file server (four cores total). This may be a more common deployment when running multiple server VMs on a single physical host, for example, because there are less cores available for each VM. The bare metal results are depicted in Figure 7(a). The first bar shows the bare metal baseline performance of 442.1 MB/s. We see in the second bar that performance drops to 331.1 MB/s when using polling. This is because the host now has only two cores, and the polling thread utilizes a disproportionate amount of CPU resources as compared to the file server. We remedied this by reducing the CPU scheduling priority of the polling thread (bar 3), and by setting the CPU affinities of the polling thread and some of the file server processes so that they share the same core (bar 4). These two changes bring the performance back to baseline performance.



(a) Bare Metal



(b) Guest

Figure 7: File server workload with 4 cores

In the guest case, depicted in Figure 7(b), the baseline (bar 1) is approximately 36% lower than the bare metal case. Bar 2 includes the HugePages and idle polling optimizations previously described, and bar 3 adds driver polling. Similar to the bare metal case, we adjusted the polling thread scheduling priority and the affinities of the relevant processes (bars 4 and 5). This brings us to results that are statistically indistinguishable from bare metal. In all cases, tuning was not difficult, and a wide range of values provided the achieved performance.

7 Related Work

Several works explored the idea of running VMs on storage controllers. The IBM DS8300 storage controller uses logical partitions (LPARs) to enable the creation of two fault-isolated and performance-isolated virtual storage systems on one physical controller [12]. Pivot3 [24] and ParaScale [22] are integrated virtualization and scale-out SAN storage platforms that are geared to data centers. Fido [8] investigated using shared memory to implement zero-copy inter-VM communication in Xen in the context of enterprise-class server appliances. Our focus is different in that we investigate external communication, zero-copy communication with the controller software, and various techniques and methods to reduce overheads caused by I/O virtualization.

Block Mason [21] used building blocks implemented in VMs to extend block storage functionality. VMware

VSA [37] pools the internal storage resources of several servers in a shared storage pool, using dedicated virtual machines running on each server.

Several works explored off-loading I/O to dedicated cores [3, 15, 16, 19]. The closest to ours is VPE [19], which adds host-side polling to KVM's virtio network stack. The VPE thread polls the network device for incoming packets and polls the guest device driver for new requests. However, the guest incurs exit overheads for interrupts and I/O completions since its driver does not poll. Dedicating cores for improving I/O performance has also been explored in TCP onloading [25, 32, 33].

There have been several works that investigated reducing interrupt overhead. The Linux kernel uses NAPI to disable interrupts of incoming packets as long as there are packets to be processed [30, 31]. A hybrid approach is to use interrupts under low load, and polling when more throughput is needed [11]. With interrupt coalescing, a single interrupt is generated for a given number of events or in a pre-defined time period [2, 27]. A series of works compared these techniques qualitatively and quantitatively [28, 29]. Rather than polling for fixed intervals or according to arrival rates, QAPolling uses the system state as determined by applications' receive queues [9]. The Polling Watchdog uses a hardware extension to trigger interrupts only when polling fails to handle a message in a timely manner [20].

ELI (ExitLess Interrupts [13]) is a recently-published software-only approach for handling interrupts within guest virtual machines *directly* and *securely*. ELI removes the host from the interrupt handling paths, thereby allowing guests to reach 97%–100% of bare-metal performance for I/O-intensive workloads.

8 Conclusions and Future Work

We have shown how to use a hypervisor to host and isolate new storage system functions with negligible runtime performance overhead. The techniques we demonstrated such as polling, dedicated cores, avoiding page lookups, etc., while not general purpose are a good fit to our usage scenario and have a significant payback.

There are several possible extensions. First, ELI [13] is a promising new approach for exitless interrupts which would remove the need to poll in the guest. We are investigating incorporating it into our system. Second, if we stay with polling, we can explore ways to better utilize the polling cores, e.g., to on-board the TCP stack to a polling core. Third, we can also benchmark these techniques when running multiple VMs. Finally, we can examine how to leverage the fact that we have virtualized the new storage function's implementation to take advantage of features such as VM migration to improve performance and availability.

Acknowledgments

Thank you to our shepherd Arkady Kanevsky and the reviewers for their helpful comments. Thanks to Zorik Machulsky and Michael Vasiliev for assisting with the hardware. The research leading to the results presented in this paper is partially supported by the European Communitys Seventh Framework Programme ([FP7/2001-2013]) under grant agreement number 248615 (IOLanes).

References

[1] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, 2006.

[2] I. Ahmad, A. Gulati, and A. Mashtizadeh. vIC: Interrupt coalescing for virtual machine storage device IO. In *USENIX Annual Technical Conf.*, 2011.

[3] N. Amit, M. Ben-Yehuda, D. Tsafrir, and A. Schuster. vIOMMU: efficient IOMMU emulation. In *USENIX Annual Technical Conf.*, 2011.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *ACM Symposium on Operating Systems Principles (SOSP)*. ACM SIGOPS, October 2003.

[5] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B. Yassour. The Turtles project: Design and implementation of nested virtualization. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, 2010.

[6] M. Ben-Yehuda, J. Mason, J. Xenidis, O. Krieger, L. van Doorn, J. Nakajima, A. Mallick, and E. Wahlig. Utilizing IOMMUs for virtualization in Linux and Xen. In *Ottawa Linux Symposium (OLS)*, July 2006.

[7] M. Ben-Yehuda, J. Xenidis, M. Ostrowski, K. Rister, A. Bruemmer, and L. van Doorn. The price of safety: Evaluating IOMMU performance. In *the 2007 Ottawa Linux Symposium*, June 2007.

[8] A. Burtsev, K. Srinivasan, P. Radhakrishnan, L. N. Bairavasundaram, K. Voruganti, and G. R. Goodson. Fido: Fast inter-virtual-machine communication for enterprise appliances. In *USENIX Annual Technical Conf.*, June 2009.

[9] X. Chang, J. Muppala, W. Kong, P. Zou, X. Li, and Z. Zheng. A queue-based adaptive polling scheme to improve system performance in gigabit ethernet networks. In *IEEE International Performance Computing and Communications Conf.*, 2007.

[10] J. Corbet. Transparent hugepages in 2.6.38. `http://lwn.net/Articles/423584/`, January 2011.

[11] C. Dovrolis, B. Thayer, and P. Ramanathan. Hip: Hybrid interrupt-polling for the network interface. *ACM SIGOPS Operating Systems Review*, 35(4):50–60, October 2001.

[12] B. Dufrasne, A. Baer, P. Klee, and D. Paulin. IBM system storage DS8000: Architecture and implementation. Technical Report SG24-6786-07, IBM, October 2009.

[13] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, D. Tsafrir, and A. Schuster. ELI: Bare-metal performance for I/O virtualization. In *Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, 2012.

[14] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux virtual machine monitor. In *the 2007 Ottawa Linux Symposium*, volume 1, June 2007.

[15] S. Kumar, H. Raj, K. Schwan, and I. Ganev. Re-architecting VMMs for multicore systems: The sidecore approach. In *Workshop on Interaction between Opearting Systems & Computer Architecture (WIOSCA)*, 2007.

[16] A. Landau, M. Ben-Yehuda, and A. Gordon. SplitX: Split guest/hypervisor execution on multi-core. In *USENIX Workshop on I/O Virtualization (WIOV)*, 2011.

[17] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, 2004.

[18] J. Liu. Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2010.

[19] J. Liu and B. Abali. Virtualization polling engine (VPE): Using dedicated cpu cores to accelerate I/O virtualization. In *the 23rd international conference on Supercomputing*, June 2009.

[20] O. Maquelin, G. R. Gao, H. H. J. Hum, K. B. Theobald, and X. Tian. Polling watchdog: Combining polling and interrupts for efficient message handling. *ACM SIGARCH Computer Architecture News*, 24(2):179–188, May 1996.

[21] D. T. Meyer, B. Cully, J. Wires, N. C. Hutchinson, and A. Warfield. Block mason. In *USENIX Workshop on I/O Virtualization (WIOV)*, 2008.

[22] ParaScale. Cloud computing and the parascale platform: An inside view to cloud storage adoption. White Paper, 2010.

[23] PCI-SIG. Single root I/O virtualization 1.1 specification, January 2010. www.pcisig.com/specifications/iov/single_root/.

[24] Pivot3. Pivot3 serverless computing technology overview. White Paper, April 2010.

[25] G. Regnier, S. Makineni, I. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong. Tcp onloading for data center servers. *IEEE Computer*, 37(11):48–48, 2004.

[26] R. Russell. virtio: Towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, July 2008.

[27] K. Salah. To coalesce or not to coalesce. *International Journal of Electronics and Communications*, 61(4):215–225, 2007.

[28] K. Salah, K. El-Badawi, and F. Haidari. Performance analysis and comparison of interrupt-handling schemes in gigabit networks. *Journal of Computer Communications*, 30(17):3425–3441, 2007.

[29] K. Salah and A. Qahtan. Implementation and experimental performance evaluation of a hybrid interrupt-handling scheme. *Journal of Computer Communications*, 32(1):179–188, 2009.

[30] J. Salim. When NAPI Comes To Town. In *Linux 2005 Conf.*, August 2005.

[31] J. H. Salim, R. Olsson, and A. Kuznetsov. Beyond Softnet. In *Anual Linux Showcase & Conf.*, 2001.

[32] L. Shalev, V. Makhervaks, Z. Machulsky, G. Biran, J. Satran, M. Ben-Yehuda, and I. Shimony. Loosely coupled TCP acceleration architecture. In *The 14th IEEE Symposium on High-Performance Interconnects*, August 2006.

[33] L. Shalev, J. Satran, E. Borovik, and M. Ben-Yehuda. IsoStack—highly efficient network processing on dedicated cores. In *USENIX Annual Technical Conf.*, June 2010.

[34] SPC. Storage Performance Council, 2007. www.storageperformance.org.

[35] J. Sugerman, G. Venkitachalam, and B. Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conf.*. USENIX Association, 2001.

[36] A. Tridgell and R. Sahlberg. DBENCH. http://dbench.samba.org/, 2011.

[37] VMware. Virtual storage appliance. http://www.vmware.com/products/datacenter-virtualization/vsphere/vsphere-storage-appliance/overview.html.

[38] P. Willmann, S. Rixner, and A. L. Cox. Protection strategies for direct access to virtualized I/O devices. In *USENIX Annual Technical Conf.*. USENIX Association, June 2008.

[39] B. Yassour, M. Ben-Yehuda, and O. Wasserman. Direct device assignment for untrusted fully-virtualized virtual machines. Technical report, IBM Research Report H-0263, 2008.

ZZFS: A hybrid device and cloud file system for spontaneous users

Michelle L. Mazurek*, Eno Thereska+, Dinan Gunawardena+, Richard Harper+, and James Scott+

*Carnegie Mellon University, Pittsburgh, PA
+Microsoft Research, Cambridge, UK

Abstract

Good execution of data placement, caching and consistency policies across a user's personal devices has always been hard. Unpredictable networks, capricious user behavior with leaving devices on or off and non-uniform energy-saving policies constantly interfere with the good intentions of a storage system's policies. This paper's contribution is to better manage these inherent uncertainties. We do so primarily by building a low-power communication channel that is available even when a device is off. This channel is mainly made possible by a novel network interface card that is carefully placed under the control of storage system protocols.

The design space can benefit existing placement policies (e.g., Cimbiosys [21], Perspective [23], Anzere [22]). It also allows for interesting new ones. We build a file system called ZZFS around a particular set of policies motivated by user studies. Its policies cater to users who interact with the file system in an *ad hoc* way — spontaneously and without pre-planning.

1 Introduction

Much work has been done in developing appropriate data placement, caching and consistency policies in the "home/personal/non-enterprise" space (e.g., see [8, 10, 16, 19, 20, 21, 22, 23, 24, 26, 28]). Good policies are crucial in maintaining good performance, reliability and availability. Unfortunately, there are many barriers that make the execution of such policies far from automatic. These barriers often stem from the unpredictability of everyday life, reflected in variable network resources, devices being off or dormant at inconvenient times, and users' time and priority given to data management.

Consider two mundane examples (Section 2 has more): In the first example, a busy mom desires to show a friend in the mall a photo that happens to be on the home computer. That same person might wish to access her personal medical file (that she does not trust the cloud for storing) from the beach while on holidays later in the week. In all likelihood she will find the above tasks impossible given that her home computer is most likely dormant or off, and she has not had time to specify any particular data replication policy among the computer and the smartphone, or hoarded the files beforehand.

The second example illustrates a consistency problem and is taken from Live Mesh's [14] mailing list. Many technology-savvy users experienced frequent conflicts with music files. A single user would listen to music on device A, then later listen to the same music on device B while A was turned off (the files were kept in peer-to-peer sync between A and B because the user did not have enough space on the cloud to store all files). Because the particular music player software updated song metadata (like play count and rating), it turns out that this is not a read-only workload. As a result, the syncing generated conflicts requiring manual resolution whenever the user switched devices. It is unfortunate that even in the absence of true multi-user concurrency, a single user can still get an inconsistent view of the system.

This paper's main contribution is to build a low-power, always-on communication channel that is available even when a device is off. The hypothesis is that this channel reduces the likelihood that a device is unreachable and thus helps the execution of data placement and consistency policies. We build this channel using new hardware and storage system protocols.

On the hardware front, we incorporate a novel network interface card (NIC) in the design of the overall storage system (Section 3.1). The NIC maintains device network access with negligible energy consumption even when the device is dormant. The NIC is able to rapidly turn on the main device if needed. The ability to turn on the main device can be thought of as Wake-on-Lan(WoL) [11] "on steroids," in that the NIC operates through any firewalls or NAT boxes, does not need to know the MAC address of the dormant device, and handles mobility across subnets. The NIC also exports to our storage system a small on-board flash storage. While the hardware part of the NIC is not a contribution of this paper, we build the storage system software around it.

We design the I/O communication channel on top of the NIC by leveraging several technical building blocks. These are not new individually, but, as we discovered, work well together to lead to a usable system. In particular, we use data placement protocols based on version histories for ensuring consistency (Section 3.3); I/O offloading [15, 29] is used to mask any performance latencies of turning on a device on a write request by using the NIC's flash storage as a versioned log/journal (Section 3.3); and users get a device-transparent view of the namespace with the metadata by default residing on the cloud. Metadata can also reside on any device with the always-on channel implemented (Section 3.2).

Fundamentally, our approach makes good use of any always-on resources, if available (such as the cloud or a home server), but also actively augments the number of always-on resources by turning any personal device with the new network interface card into an always-on resource. Perhaps subtly, however, it turns out that having a few extra always-on resources allows for interesting data placement policies that were not possible before. We explore these through building a file system called ZZFS. We chose to implement a unique set of data placement and consistency policies that cater mostly to spontaneous users (Section 4). These policies were partially influenced by qualitative user research. However, other policies (e.g., Cimbiosys [21], Perspective [23], Anzere [22]) would equally benefit.

2 Background on the problem

Users often have access to a set of devices with storage capabilities, such as desktops, laptops, tablets, smartphones and data center/cloud storage. *Data placement* policies revolve around deciding which user's data or files go onto which device. Often, a data placement policy indicates that the same file should be placed on multiple devices (e.g., for better reliability, availability and performance from caching). *Consistency* policies revolve around ways of keeping the multiple file replicas in sync as to provide the abstraction of a single file to users. We illustrate problems related to the execution of these policies through three simple examples, that reflect policies taken from some recent related work.

Example 1: Two replicas of a file: This example defines the terminology and thus is slightly longer than the subsequent two. Systems like Perspective [23], Cimbiosys [21] and Anzere [22], allow a photographer to say "keep all my photos replicated on my work machine and tablet." Imagine a user U and a photo file F. It is very likely that when U edits F from the work machine, the tablet is dormant so the changes do not immediately propagate to the tablet. Typical implementations of this policy make use of a transaction log L that keeps track of the changes U makes on the work machine. The log is later replayed on the tablet to maintain consistency.

When the photographer later on moves to work on the tablet, the log will still be on the now-dormant work machine. Thus, the tablet is not able to replay the log. The user has two options, neither which leads to great satisfaction with the system: option 1 is for the user to manually turn on the work machine and wait until all the data is consistent. This option is implicitly assumed in Perspective, for example. Option 1 may be out of reach for non tech-savvy users who just want to get on with their work and do not understand they have to wait ("for how long?") for consistency to catch up.

Option 2 is to continue working on the stale copy of F on the tablet, keep a separate transaction log L_2 of the work in the tablet, and then later on, when both machines happen to be up at the same time, have a way to reconcile L and L_2. In the best case, the copies can be reconciled automatically (e.g., the user is working on two different parts of the photo that can be just merged). In the worst case, manual conflict resolution is required. Option 2 is in fact the only option if there is truly no other way the devices can communicate with one another (e.g., if the user is on a plane with the tablet and with no network connectivity). However, it seems wasteful human effort that the user has to resort to this option even when the network bandwidth in many places (e.g., within the home, or work) would be perfectly adequate for automatic peer-to-peer sync, if only the devices were on.

Example 2: Device transparency: Several systems advocate *device transparency*, where the namespace reflects ones' files and data, not the device where they reside. Eyo, for example, allows a user to list from *any* device the metadata (e.g., name) of all files, residing in *all* subscribed devices [26]. We like the idea of the metadata being always available, but want to help further by satisfying the user's data needs as well. Imagine a user U having the names of all her documents, photos and videos, displayed on her tablet. When U meets a friend in the mall, she wishes to show her a short video from a birthday party. The video happens to physically reside on her home computer (although the metadata is on the tablet). There is reasonable 3G bandwidth to stream the video, but the home computer is dormant. The user knows the video exists, but cannot access it.

Example 3: Cloud storage: Having sufficient storage space to store *all* user data in the cloud with fast network connectivity to access it seems technically likely in the next few years (perhaps sooner in Silicon Valley). However, any consideration of data placement must include

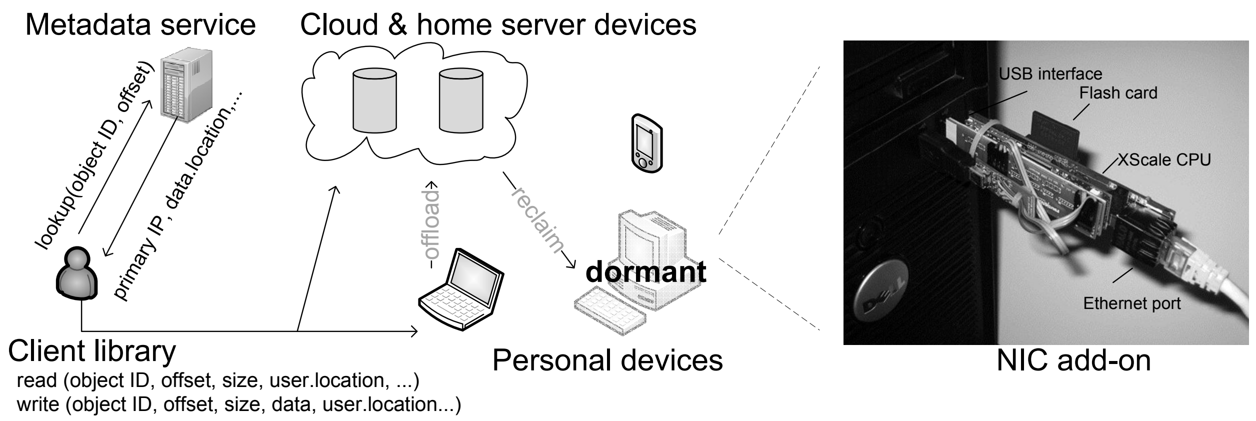

Metadata service
Cloud & home server devices
lookup(object ID, offset)
primary IP, data.location,...
offload
reclaim
dormant
USB interface
Flash card
XScale CPU
Ethernet port
Client library
read (object ID, offset, size, user.location, ...)
write (object ID, offset, size, data, user.location...)
Personal devices
NIC add-on

Figure 1: Storage system architecture, basic interfaces and Somniloquy hardware in action.

human factors as well as technology and cost trends. Human factors include, among others, trust in the cloud and desire to possess, know and control where one's data is located. Section 4 describes qualitative user studies we did in the context of this paper. From those studies, we believe that devices will continue to be places where users store some of their data. As such, we fully embrace the cloud as another place to store data, and we let users ultimately decide how to use that place. We do not second-guess them or force them to automatically place everything on the cloud. Internally, the system makes good use of available cloud space (e.g., for storing metadata — Section 3.2, and versioned logs — Section 3.3).

On the technical front, our system helps users who might have slow network connections to the cloud. Imagine a scenario in which a user decides to store a substantial amount of his data on the cloud. A user editing an article and compiling code while traveling benefits from the device's cache to batch writes before sending them to the cloud. When the user returns home and wants to continue working on the data from his home PC, he finds the PC's state is stale and incurs large performance penalties until the state is refreshed. A good cache placement policy would automatically hoard the user's working set to the home cache before the user would need to use it. Such a policy is hampered, however, because the home PC is likely dormant before the user arrives.

Intuition on how this paper helps: This paper is about enabling a satisfying execution of a user's data placement, caching and consistency policies given the likelihood that devices they rely on are dormant. One way our system will help the situation in Example 1 is by allowing peer-to-peer sync policies to work by turning devices on and off rapidly and automatically. If peer-to-peer sync would not be advisable (e.g., because of battery considerations), the system temporarily offloads the transaction log *L* onto the cloud. In Example 2, the system will continue to present a device-transparent view of metadata, and will rapidly turn on the home computer to get the data to the user. In Example 3, either peer-to-peer or cloud-to-device cache syncing will be enabled by turning the devices whose caches need refreshing on.

3 Design

Figure 1 shows several building blocks of the storage system. First, storage-capable devices strive to always maintain a low-power communication channel through a new low-power network card. Second, a metadata service maintains a unified namespace, encompassing any available storage space on devices, cloud and any home servers. Third, an I/O director, in cooperation with the metadata service and the new communication channel, manages the I/O flow through the system.

3.1 Maintaining network awareness

Data placement and consistency protocols are helped if devices maintain an always-on communication channel, even when dormant or off. Of course, such a channel should consume minimal power. We chose to use a new network interface card, called Somniloquy, that is designed to support operation of network-facing services while a device is dormant. Figure 1 shows it operating with one of our desktops. Somniloquy was first described by Agrawal et al. [2] in the context of reducing PC energy usage. The hardware is not a contribution of this paper. This paper reports on Somniloquy's role and integration into a distributed personal storage system.

Somniloquy consumes between one and two orders of magnitude less power than a PC in idle state. Somniloquy exports a 5 Mbps Ethernet or Wireless interface (Figure 1 shows a prototype with the Ethernet interface) and a few GB of flash storage. Somniloquy runs an embedded distribution of Linux on a low power 400 MHz XScale processor. The embedded OS supports a full TCP/IP stack, as well as DHCP and serial port communication. Power consumption ranges between 290 mW for an idle wireless interface, 1073 mW for the idle Ethernet interface, and 1675 mW when writing to flash [2].

Somniloquy allows a dormant device to remain responsive to the network. The NIC can continue to communicate using the same IP address as the dormant device. Somniloquy is more appropriate than Wake-on-LAN (WoL) [11] for mobile storage devices, because it operates through firewalls and NAT boxes, and it handles mobility across subnets. The on-board processor maintains contact with a DNS server to preserve the hostname-to-IP address mapping, performs basic networking tasks, and does I/O to its local flash card.

Does the new NIC make the overall system less secure? Our experience is incomplete. Logically, the system is running the same storage service as before. However, because parts of that service now run on the NIC's processor, the attack surface on the system as a whole has increased. Also, while modern processors have additional security features such as execute-disable bits to prevent buffer overflows, our low power processor does not support these features yet. Denial-of-service attacks might result in drained batteries. To partially mitigate these problems we force the NIC to only listen on one port (5124) that belongs to the storage service. Further, we require the main device and the NIC's processor to be on the same administrative domain.

Somniloquy is the hardware part of the solution, but it is insufficient without the storage and file system software. Here we give intuition on how the I/O director (Section 3.3) will use Somniloquy for two common operations: reads and writes. A read to a file on a Somniloquy-enabled storage device incurs a worst-case latency when the request arrives just as the device is going into standby. Somniloquy will wake up the device and the latency is at least *standby* + *resume* time. Table 1 shows some measurements to understand this worst-case penalty. Future devices are likely to have faster standby and resume times. Writes do not have a similar latency penalty. The I/O director can temporarily offload data to Somniloquy's flash card, or nearby storage-capable resources (such as the cloud) if these are available.

Summary, limitations and alternatives: We design to allow devices to maintain network awareness even when dormant. Our specific way of enabling the goal is to introduce new NIC hardware to each device. Agrawal et al. [2] describes why Somniloquy is more appropriate than several other hardware-based alternatives (e.g., Turducken [25]) and we do not list those alternatives further here. An assumption we make is that it is cost effective to augment devices with a smarter network interface card. Further, we assume the NIC would not drastically change the failure characteristics of the device. These assumptions might turn out to be a limitation of our approach, depending on the economics of producing a device and its failure characteristics. Another limitation is a lack of evaluation of Somniloquy with tablets or smartphones. Currently the driver works for Windows Vista/7 only, which limits the experiments in Section 5 to laptops and desktops. Currently, the NIC can only wake up devices that are placed into standby, and are not fully off.

Device	Standby(s)	Resume(s)
Lenovo x61 (Win7)	3.8	2.6
Dell T3500 (Win7)	8.7	7.2
HP Pavillon (XP)	4.9	10.25
Macbook Pro (OSX 10.6.8)	1	2
Ubuntu 11.10	11	4.5

Table 1: Example suspend and resume times for commodity devices. The device is first rebooted to clear previous state then it is put into standby followed by a resume. Section 5.2 shows more realistic end-to-end measurements using the Dell T3500 device.

A software-based alternative would be to maintain device network awareness by encapsulating a device in a virtual machine abstraction and then making sure the virtual machine (VM) is always accessible. SleepServer, for example, migrates a device's VMs to an always-on server before the physical device goes dormant [3]. This alternative might be more appropriate in enterprise environments where VMs are used and dedicated always-on servers are available, rather than for personal devices.

3.2 Metadata service

The metadata service maintains a mapping among an object/file ID, the devices that object is stored onto, and the replication policy used. The MDS uses a flat object-based API by default, where each object ID is an opaque 128-bit string. The metadata service (MDS) is a logical component, and it can reside on any device or server. The metadata service might be replicated for availability. Consensus among replicated services could be maintained through the Paxos protocol [22]. Furthermore, the data belonging to the service might be replicated for reliability, or cached on devices for performance. Data consistency needs to be maintained across the replicas.

The low-power communication channel in Section 3.1 helps with MDS availability and reliability in the following way. If the service is replicated among devices for availability, Somniloquy wakes up dormant devices that need to participate in the consensus protocol. If the data belonging to the MDS is replicated, the I/O director strives to maintain strong consistency through a range of techniques described in Section 3.3. A reasonable default for home users is to have a single instance of the metadata service run on a cloud server with content replication factor of 1, i.e., instead of being replicated, the metadata content is cached on all devices (this is what our file system implementation in Section 4 does). The metadata content can be cached on all devices since its size is usually small (Section 5.5).

The client library caches a file's metadata when a file is created and pulls metadata updates from the metadata service when accesses to a file fail. The latter could happen either because the file has moved or it has been deleted, the access control policy denies access, or the device has failed. A client's library synchronously updates the MDS when metadata changes. Those updates could be subsequently pushed by the metadata service to other devices caching the metadata (the push could be lazy, e.g., daily, or could happen as soon as the change occurs). For the common case when a device is dormant, Somniloquy could wake up the device (or absorb the writes in its flash card temporarily) to update its cache. A client might choose to pull the latest metadata explicitly (e.g., through a Refresh button), rather than using the push model. While the design supports both models, we believe a hybrid pull and lazy push model is a reasonable default for home users.

Our design requires storage devices to be explicitly registered with the MDS. If a device is removed from the system, either because it has permanently failed or because a newer device has been bought that replaces it, a user needs to explicitly de-register the old device and register the new device with the MDS. The metadata service initiates daily heartbeats to user devices to detect permanent failures and to lazily refresh a device's metadata cache. A heartbeat wakes up a dormant device. A device is automatically rebuilt after the user triggers the rebuild process.

Summary, limitations and alternatives: The novel aspect of our metadata service is that the execution of both metadata service consensus (for availability) and metadata replication consistency protocols (for reliability and performance through caching) is helped by the ability to turn participating devices on and off transparently. The design allows for several consensus and consistency options. However, by default the MDS resides on the cloud and its content is cached on all devices. The implicit assumption for this default is that the user will have at least (>56 Kbps) broadband connectivity at home or work and some weak 3G connectivity when mobile. Further, we assumed a few hundreds of MB of storage space at a cloud provider. We believe this is a weak assumption, but, even in the absence of cloud space, the metadata service and data could still reside on any device that incorporates Somniloquy.

3.3 I/O director

The I/O director is the third building block of our design. Its goal is to be versatile, allowing for a range of data placement and consistency policies. Uniquely to our system, the I/O director has new options for data movement. It can choose either to wake up a device to make reads or writes, or to temporarily use the flash storage provided by Somniloquy; it can also opportunistically use other storage resources to mask performance latencies and maintain the always-on communication channel.

The operations of the I/O director are best understood through Figure 2, which shows a client, a metadata service (MDS) and two devices D_1 and D_2. The data is replicated on both devices with a particular primary-based concurrency control mechanism to serialize concurrent requests. In this example, each replicated file has one replica that is assigned the primary role. Figure 2 shows some common paths for read and write requests. Reads, in the default case, are serviced by the primary for an object, as seen in Figure 2(a). When all devices are dormant and a read or write request arrives, Somniloquy resumes the device and hands it the request as shown in Figure 2(b) for reads and Figure 2(d) for writes, respectively. Writes are sent to the primary, which serializes them to the other replicas of the object as in Figure 2(d).

When objects are replicated and a device goes into a controlled standby, the metadata service receives an RPC indicating that, as seen in Figure 2(c). This is an optimization to give the MDS the chance to proactively assign the primary role away from that device to devices that are on. As might be expected, transferring the primary role does not involve data movement, just network RPCs to inform devices of the new assignment. A client's metadata cache might be stale with the old primary information, so a read will initially go to the dormant device. However, the device is not turned on, since the primary does not reside there. Instead, the client times out, which triggers an MDS lookup and cache refresh. The read then proceeds to the device with the primary, which happens to be on in this example.

The I/O director implements I/O offloading [15, 29] to mask large write latencies and to implement the logging subsystem. The logging subsystem gives the abstraction of a single virtual log to the whole distributed storage system. The actual log might reside on any storage device. Its size is limited by the size of cloud space, plus NIC flash space, plus all free hard drive space across all devices. Figure 2(e) shows offloading to the log (Section 5 evaluates the case when the log physically resides on a nearby device). Remember that if parts of the log are on the dormant device's hard drive, that device can be woken up as needed to access the log. Data is eventually reclaimed at the expected device at appropriate times,

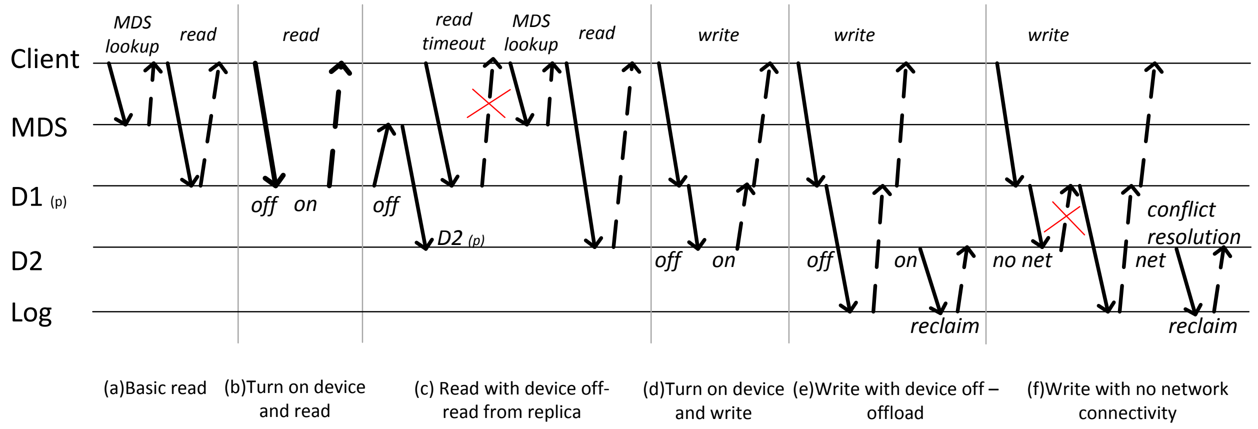

MDS lookup
read
read
read timeout
MDS lookup
read
write
write
write
Client
MDS
D1 (p)
D2
Log
off
on
off
D2 (p)
off
on
off
on
reclaim
no net
net
conflict resolution
reclaim
(a)Basic read
(b)Turn on device and read
(c) Read with device off- read from replica
(d)Turn on device and write
(e)Write with device off – offload
(f)Write with no network connectivity

Figure 2: Read and write protocols and common cases. *D* stands for device and *p* indicates that the primary for the file being accessed is on that device. "Off" and "on" indicate whether the device is dormant or not.

e.g., when the device is not in use.

The system is optimized for the common case when there is some network connectivity among devices and the cloud. If that is not the case, e.g., when the user is on a plane without network access, the system will temporarily offload all user writes to the log, and the log will have to physically reside with the user's device locally. When the user gains network connectivity, all participating devices will have to eventually reclaim data from the log and do standard conflict resolution (e.g., as in Bayou [28]), as illustrated in Figure 2(f). Our work does not add anything novel to this scenario's logic, but our implementation makes use of the existing logging infrastructure to keep track of write versions.

A user can move the file to a new device, and can change its replication policy any time. When any of these options happen, our system allows continuous access to the file. Any new writes to the file are offloaded to the versioned log. The I/O director logic maintains the necessary bookkeeping to identify the location of the latest version of a file. The location could be the old location, or the log, depending on whether the file has seen any new writes while being transferred or not. Once the file has moved to the new location, reclaim is triggered to copy any bytes that might have changed.

Summary, limitations and alternatives: The novel aspect of the I/O director is that it has new options for data movement. It can also choose to turn on a dormant device. The I/O director is optimized for an increasingly-common case of at least basic network connectivity among storage devices. It reverts to well-known conflict resolution techniques otherwise.

We currently use I/O offloading techniques [15, 29] to augment the base file system (which is not versioned) with a versioned file system partition. Ursa Minor's techniques for data placement versatility [1] are a good alternative in case the underlying file system is already versioned. For example, Ursa Minor uses backpointers when changing data replication while maintaining data availability. Also, advanced data encoding policies (e.g., the use of erasure codes), and other concurrency control methods (e.g., based on quorums) could equally benefit from our always-on communication channel.

3.4 Interaction with energy policies

As remarked above, Somniloquy consumes more than an order of magnitude less energy than an idle device while maintaining network awareness. The default interaction with energy policies is simple. A read overrides the energy policy and wakes up the device. Writes are fully buffered in the NIC's card or cloud before waking up the device. These policies are similar to the ones offered by BlueFS [16], in that they actively engineer and divert the traffic to the right device, but we have more resources available, in the form of the NIC's flash card or cloud.

Because the NIC runs a capable operating system, more complex energy policies can be encoded as part of the NIC processing. For example, BlueFS reduces energy usage by reading data from the device that will use the least amount of energy. That policy could be slightly modified to take into account the device turn on time, if the device is dormant. Furthermore, the storage system could determine whether to wake up a device or not as a function of whether the device is plugged in or running on batteries. Also, a more advanced standby strategy might predict future access patterns and prevent the computer from going into standby. Currently, our devices use simple idle time-based policies, like the ones implemented on Windows.

4 ZZFS: a file system artifact

Perhaps surprisingly, having a few extra always-on resources allows for interesting data placement policies that were not possible before. We explore these through building a file system called ZZFS. We chose to implement a unique set of data placement and consistency policies that cater mostly to spontaneous data accesses.

4.1 Design rationale

The design rationale for ZZFS is indirectly influenced by data from qualitative user research, comments on mailing lists of popular sync tools like Live Mesh [14] and Dropbox [4], and our desire to explore new policies. ZZFS's policies are different from those of say, Cimbiosys [21] or Perspective [23], but not necessarily "better" or appropriate in all cases.

Data from sync programs: To understand how users perceive consistency and conflict problems and how they rate them in fix-priority when compared to performance problems we collected and analyzed user feedback for Live Mesh [14] and Dropbox [4], two popular rsync tools. They serve as a rather coarse proxy for understanding consistency in the absence of a distributed file system. Feedback from the sync programs is heavily biased toward early adopters and technology experts, of course, but it is nevertheless helpful if only because of its volume (thousands of messages on public forum boards). Example 1 in Section 2 was influenced by this data.

Qualitative studies: Our first qualitative study helped us understand how people understand, organize, store, and access their content across different devices. The users for the qualitative studies were picked at random by a third-party company that specializes in user studies. We performed "guerrilla" (street) interviews with six people. We visited two family homes and we then invited two different families to a conference room (provided by the third-party company so that our identities would remain unknown to avoid perception bias) to further discuss concepts through storyboards. The raw data we collected is available upon request, but we have not put it in paper form yet. In parallel, we conducted a second, larger-scale study on issues around data possession [17].

How the data influenced us: This research influenced us to try harder to cater to the character of data access and device management displayed by ordinary (i.e., non technical) users. We interpret the data as suggesting that syncing and replication policies are compromised by the ways users store data, their *ad hoc* access of networks, and the priority given to social and economic matters of data management.

By default, ZZFS caters to spontaneous users with no data placement policies specified at all by default. No user effort is required to pre-organize data on devices (by hoarding, syncing, etc.) Data by default remains on the device where the user chose to first create it, with a replication factor of 1. Users showed a greater concern for and doubts about transferring data between devices than device failure. This could be interpreted as similar to Marshall's observation that only 5% of data loss is due to a device failure [12].

Whenever a file needs to be accessed, the device it is on is asked to provide access to that file. If the device is dormant, the device is woken up through the I/O director and the network-aware part of the device. For more advanced users who worry more about device failure and thus specify a higher replication factor for files, ZZFS strives to reduce the time it takes to reach consistency among replicas by data offloading and by waking up devices as described in Section 3.3.

We found that users made deliberate and intelligent decisions about wanting to silo their data on different devices and the cloud. From both user studies, we believe that devices will continue to be places where users store their data. Any consideration of data placement must consider human factors as well as technology and cost trends. Human factors include, among others, trust in the cloud and desire to possess, know and control where one's data is located. Furthermore, different devices have unique affordances [6] and properties (e.g., screen size, capacity, weight, security, price, performance, etc.). Users seem capable of understanding those affordances, and ZZFS does not second guess. Data movement is incurred only when a user explicitly chooses to do so.

4.2 Implementation details and status

We have implemented most of the design space described in Section 3. ZZFS is a distributed file system that results from picking a set of policies. It is implemented at user-level in *C*. ZZFS supports devices whose local file system can be NTFS or FAT. ZZFS has implemented per-object replication and allows for in-place overwrites of arbitrary byte ranges within an object. Concurrent accesses to a file are serialized through a primary. ZZFS's namespace is flat and it does not have folders, however it maintains collections of files through a *relate()* call.

The current implementation addresses a limited set of security concerns. Data and network RPCs can be encrypted (but are not by default). Each object has an access control list that specifies which user can access that object and from what device. We are actively doing research in what security means for home users [13].

In addition to simple benchmarks that directly access ZZFS through a client library, we run unmodified, legacy applications (e.g., MS Office, iTunes, Notepad, etc.) for demoing and real usage. We do so by mounting ZZFS as a block-device through the use of a WebDav service [31]. This technique required us to detour the WebDav service to use our APIs [9]. WebDav file semantics are different from NTFS semantics and often lead to performance inefficiencies (e.g., any time a change is made to a file, WebDav forces the whole file to be sent through the network). The following calls are detoured to use ZZFS's calls: *CreateFile()*, *FindFirstFile()*, *FindNextFile()*, *ReadFile()*, *WriteFile()*, *GetFileAttributes()* and *DeleteFile()*. The interface currently is Windows Explorer. A more appropriate interface for a distributed file system is work-in-progress.

ZZFS is robust. We are using it daily as a secondary partition to store non-critical files. When it crashes, it usually does so because of the NIC's device driver. The driver issues will be resolved over time and were not our primary focus for this paper. However, we are working toward having ZZFS as a primary partition for all files.

5 Evaluation

First, we measure how ZZFS performs and locate its bottlenecks. Second, through a series of real scenarios, we measure latencies and penalties associated with the always-on communication channel. This is an evaluation of the underlying storage system and also of ZZFS's policies. Third, we provide analytical bounds for performance for a range of workload and device characteristics. Fourth, we examine metadata scalability.

5.1 Exposing throughput bottlenecks

This section focuses on **throughput**. The other sections will focus on latency. We compare our system against local file access through the NTFS file system. This is the only time we will use a set of homogeneous devices (obviously not realistic for personal devices), because it is simpler for revealing certain types of bottlenecks. The devices are three HP servers, each with a dual-core Intel Xeon 3 GHz processor and 1 GB of RAM. The disk in each device is a 15 KRPM Seagate Cheetah SCSI disk. The devices have a 1 Gbps NIC. All reads and writes are unbuffered, i.e., we do not make use of the RAM.

First, we measure peak bandwidth and IOPS (I/Os per second) from a single device ("Read.1" and "Write.1" in Figure 3). Bandwidth is measured in MB/s using 64 KB sequential reads and writes to a preallocated 2 GB file. IOPS are measured by sending 10,000 random-access 4 KB IOs to the device with 64 requests outstanding at a time. Figure 3 shows the average from 5 results (the variance is negligible). Average local streaming NTFS performance (not shown in graph) is 85 MB/s for reads and writes and 390 IOPS for reads and 270 IOPS for writes; hence, ZZFS adds less than 8% overhead.

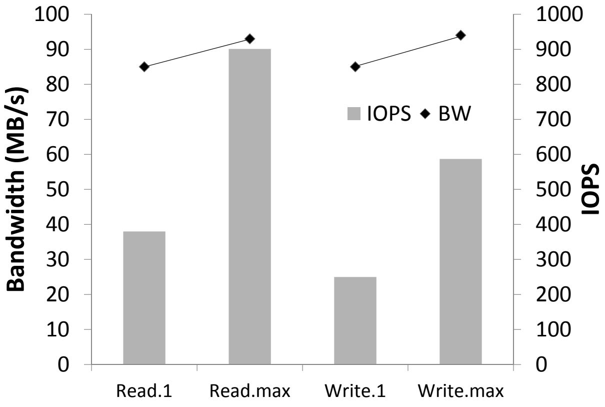


Figure 3: Baseline bandwidth and IOPS.

Second, we measure maximum bandwidth and IOPS from all three devices to understand performance scalability ("Read.max" and "Write.max" in Figure 3). Three clients pick one random 2 GB file to read or write to, out of a total of 10 available files. Each file is replicated 3-way. If all clients pick the same file, accesses still go to disk since buffering is disabled. Figure 3 shows the results. As expected from 3-way replication, the saturated write bandwidth is similar to the bandwidth from a single device. Saturated read bandwidth is about a third of the ideal because requests from all three clients interfere with one another. This problem exists in many storage systems because of a lack of performance isolation [30]. Saturated IOPS from all devices is close to the ideal of 3x the IOPS from a single device.

Overall, these results show that our system performs reasonably well with respect to throughput. Optimizations are still required, however, especially with respect to reducing CPU utilization. CPU utilization in the saturated cases was close to 100%, mostly due to unnecessary memory copies.

5.2 I/O director

This section focuses on read and write **latencies** resulting from the always-on channel. We have real measurements from a home wireless network. We start by illustrating the performance asymmetry between reads and writes. The first workload is an I/O trace replay mimicking a user listening to music. We use trace replay to just focus on I/O latencies and skip the time when the user listens to music and no I/O activity is incurred. Half of the music is on his laptop, half on the desktop and the setting is in "shuffle mode" (i.e., uniform distribution of accesses to files). The music files are not replicated. The desktop (Dell T3500 in Table 1) is on a 100 Mbps LAN and the laptop (Lenovo x61) is on a 56 Mbps wireless LAN. The Somniloquy NIC is attached to the desktop. The MDS

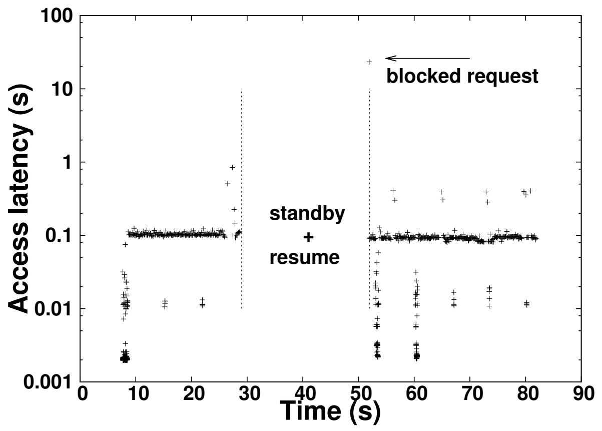


Figure 4: A scatter plot over time for a client's requests. Reads latencies are [mean=0.09 s, 99^{th}=0.36 s, worst=23.3 s]. Write latencies are [mean=0.014 s, 99^{th}=0.022 s, worst=0.058 s]. There are several performance "bands" for local reads (0.001-0.01 *s*), remote writes (0.05-0.1 *s*) and remote reads (0.05-1 *s*).

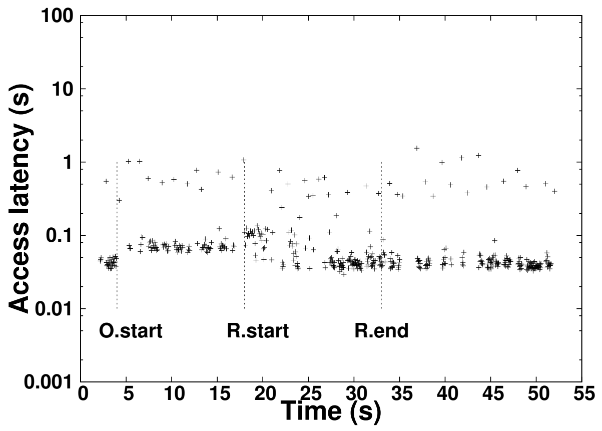


Figure 5: A scatter plot over time for a client's write requests. *O.start* annotates the time the second device enters standby, and thus offloading begins to a third device. *R.start* annotates the time when the second device resumes and reclaim starts (offloading thus ends). *R.end* annotates the time when all data has been reclaimed. Write latencies are [mean=0.1 s, 99^{th}=1 s, worst=1.5 s].

resides on the desktop, but all metadata is fully cached on the laptop as well. The music program issues 64 KB reads to completely read a music file, then, after the user has finished listening, a database is updated with a small write of 4 KB containing ratings and play count updates. The database resides on the desktop and is not replicated. Hence, although this is a common workload, it is quite complex and has both reads and writes. The user simply wants to listen to music without worrying where the music files and database are located.

Figure 4 shows a scatter plot (and latency distribution in the caption) of the worst-case scenario when request to read a music file comes just as the desktop is starting to go into standby. Somniloquy intercepts the read request and signals the computer to wake up. The time it takes the computer to accept the request is 23.3 s (*standby time + resume time*) and is illustrated in the scatter plot in the figure. In practice, prefetching the next song would be sufficient not to notice any blocking; however, when prefetching is not possible, this serves as a worst-case illustration. We note that the desktop is rather old, and if using a newer device (e.g., the Macbook Pro in Table 1) the worst case latency would be around 4 seconds.

Figure 5 illustrates that writes do not suffer from this worst case scenario. The workload in this scenario is a trace replay of I/O activity mimicking a user sending 64 KB writes to a document from the laptop. The user uses 2-way replication for those files, with the second replica kept on the desktop. Both laptop and desktop are on the wired LAN. Similar to the previous case, the desktop has gone abruptly into standby. However, there is a second laptop nearby that is on, and the I/O director temporarily offloads the writes onto that laptop (other options for the offload location are Somniloquy's flash card or the cloud). This way, 2-way, synchronous replication is always maintained. When the desktop comes out of standby, the data on the third laptop is reclaimed. Reclaim does not lead to a noticeable latency increase. The figure shows a slight increase in latency during data offload since the second laptop is on the wireless LAN. A handful of requests experience high latencies throughout the experiment. We believe these are due to the performance of the wireless router. Note that writes in this experiment are slower than in Figure 4 because of larger write request sizes (64 KB vs. 4 KB) and 2-way replication vs. no replication.

We compare our system against simple *ping* and average disk latencies, i.e., we set a relatively high bar to compare against. We measured a minimum of 0.06 s ping latency for 64 KB[1], 0.005 s for 4 KB sizes and the disk's average latency is 0.015 s (these are slow SATA disks, not the fast SCSI disks used in the previous section). Hence, an end-to-end read request (and ack) should take on average 0.075 s and an end-to-end write request (and ack) should take on average 0.02 s[2]. Looking at the performance "bands" in Figure 4, we see that local read latency and remote write latency is very good, while remote read latency is 33% slower than ideal. We have

[1]Exact size is 65500 B, the maximum ping size.

[2]Although read requests are sequential, the disk head incurs at least a full disk rotation before receiving the next request, since the requests are sent one at a time. Also, experienced disk average latencies are sometimes better than the above theoretical value because our disk is not full and the files are on its outer tracks.

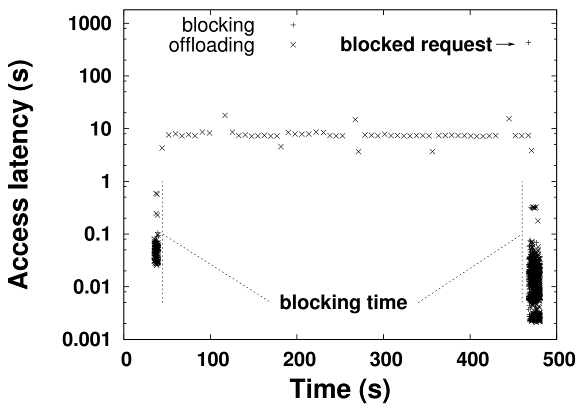


Figure 6: A scatter plot over time showing the effects of moving a file on concurrent operations on that file. Without offloading, the concurrent workload blocks; with offloading, the concurrent workload makes progress. When offloading, read latencies are [mean=0.7 s, 99th=8.6 s, worst=17 s]. Write latencies are [mean=0.5 s, 99th=7.3 s, worst=14 s].

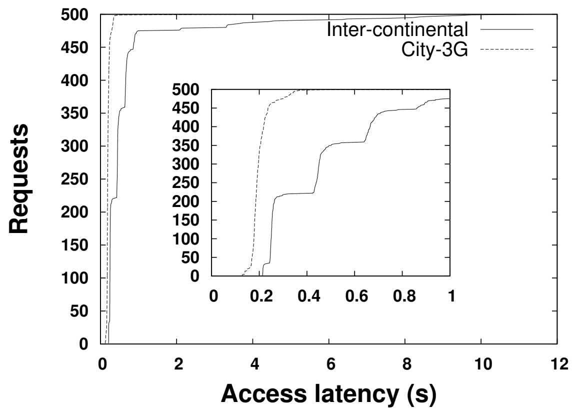


Figure 7: A CDF plot for a client's read and write requests over a 3G city-wide network and intercontinental network. For the 3G network, read latencies are [mean=0.21 s, 99th=0.35 s, worst=3.39 s]. Write latencies are [mean=0.17 s, 99th=0.3 s, worst=0.3 s]. For the intercontinental network read latencies are [mean=0.7 s, 99th=8.2 s, worst=11 s]. Write latencies are [mean=0.2 s, 99th=0.4 s, worst=0.4 s].

started collecting detailed performance profiles, but we note that the delay is unnoticeable to the applications.

File move: The next experiment demonstrates how moving an object affects performance of concurrent operations on that object. As discussed in Section 3.3, instead of locking the file for the duration of the move, the I/O director offloads any new writes to the file while the copy is in progress. In this experiment, we move a 1 GB file from one device to another while simultaneously running a series of 64 KB read and write (with a 1:1 read:write ratio) operations on that object. Figure 6 shows that, with offloading turned off, the read and write operations must block until the data move is complete; with offloading turned on and another laptop temporarily absorbing new writes, these operations make progress. The devices are limited by the 56 Mbps wireless LAN, and the network is saturated during the file move, hence access performance during that time is slow (around 10 s). We believe this is better than blocking for more than 400 s (the latency of "blocked request" in the figure). Note that after the move completes, performance improves because the client is co-located with the device the file is moved onto.

5.3 ZZFS's placement policy

Next, we measure ZZFS's performance in a 3G city-wide network and an intercontinental network. We look at the performance resulting from the simple policy of leaving data on the device it was first created. We illustrate the performance of our system when a user on the move is accessing music files stored on the home desktop. Unlike the music scenario above, the client has no music files or metadata cached on the laptop and always reads and writes to the home desktop. Access sizes are the same as before (64 KB reads and 4 KB writes).

First, when the user is on a city-wide 3G network, she is connected to the Internet through a ZTE MF112 mobile broadband device connected to her laptop. Figure 7 shows the latency results. The first request incurs a first-time setup cost from the 3G provider, which is also the worst-case latency (we do not know what the provider is doing; subsequent runs do not incur this penalty, but we show the worst case). We measured a minimum of 0.23 s ping latency for 64 KB sizes, 0.13 s for 4 KB sizes in this environment, and ZZFS's overhead is comparable. The latency is good-enough for listening to music.

Second, when the user is on the west coast of the US (Redmond, Washington) she is connected to the Internet through a 56 Mbps wireless LAN. The location of the music files is on a desktop in Cambridge, UK. Figure 7 shows the results. We measured a minimum of 0.25 s ping latency for 64 KB sizes, 0.19 s for 4 KB sizes in this environment. ZZFS's write overhead is comparable, but its average read latency is 60% higher than ping. We believe this is due to the unpredictable nature of the intercontinental network. Nevertheless, the user does not perceive any noticeable delay once the music starts.

A takeaway message from this section is that ZZFS's performance is good enough in all cases for the applications involved. Data is never cached in these experiments, so we expect even better performance in practice.

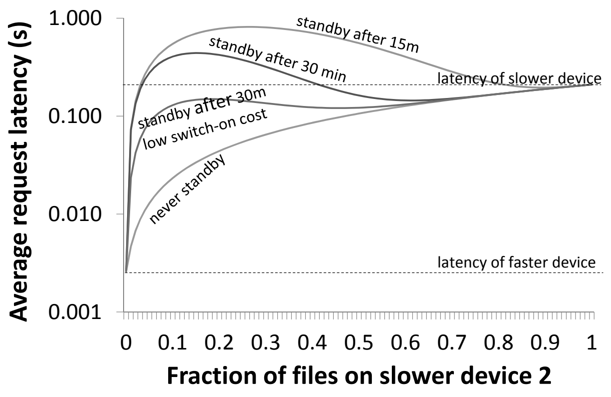


Figure 8: Latency tradeoffs for a client's read requests.

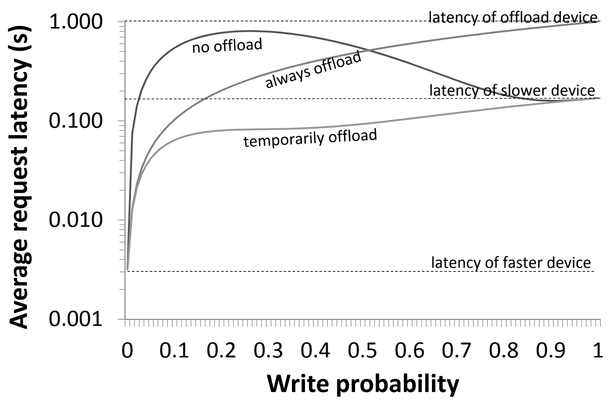


Figure 9: Latency tradeoffs for a client's requests when data is replicated on both devices.

5.4 Sensitivity to parameters

This section reexamines the above scenarios and others *analytically* while changing several tunable parameters.

In the next analysis, we revisit the music scenario. We still have two devices D_1 and D_2. First, we vary the amount of idle time I before D_1 enters standby (D_2 never enters standby since it is the device with the music player). Without loss of generality, we assume D_1's average access latency when D_1 is on, $L_1{}^{ON}$, is slower than D_2's average access latency L_2 (e.g., D_1 could be on the 3G network). $L_1{}^{STDBY}$ is the average access latency when D_1 is on standby. It is the time to resume the device plus $L_1{}^{ON}$.

Second, we vary the fraction of files p_1 that reside on the slower device ($p_2 = 1 - p_1$). For example, if D_1 enters into standby after $I = 15$ idle minutes and each song is on average $M = 5$ minutes in length, D_1 will enter standby if at least $\lfloor I/M \rfloor = 3$ consecutive songs are played from D_2 (with no loss of generality, we assume the writes go to a database also on D_2 this time, otherwise D_1 will never enter standby). Figure 8 shows the expected average latency given by:

$$E[L] = E[L|D_1 = ON]p\{D_1 = ON\} + \\ E[L|D_1 = STDBY]p\{D_1 = STDBY\} \quad (1)$$

The above equation further expands to $E[L] = (p_1 L_1{}^{ON} + p_2 L_2)p\{D_1 = ON\} + (p_1 L_1{}^{STDBY} + p_2 L_2)p\{D_1 = STDBY\}$. The analysis assumes a user is forever listening to songs, and this graph shows the long-running latency of accesses. All the lines assume the switch-on times of the Dell T3500, except for the low switch-on cost line that is the Mac.

We make several observations. In both extremes, where all files accessed are on D_2 or all files are on D_1, the latency is simply that of D_2 or D_1 respectively. If a device goes into standby, the worst latency tends to happen when the user accesses it infrequently, thus giving it time to standby and then resuming it.

The next analysis examines the impact of the read:write ratio of the workload. 2-way replication is used, and the same arguments are made about standby. The difference is that, in this case, D_1 enters standby if there are consecutive reads on D_2 (a write would wake up D_1 since it needs to be mirrored there.) Without loss of generality, we assume a read or write comes every 5 minutes and D_1 enters standby after $I = 15$ minutes. We illustrate the impact of turning on the device vs. always offloading (unrealistic in practice) vs. temporarily offloading while the device switches on. We assume without loss of generality that data is offloaded to a slow device, e.g., a data center.

Figure 9 shows the expected average latency $E[L]$ (a similar formula to the previous example is used, but the standby latency is the offload latency). We make several observations. For an all-read workload all files are read from D_2 (faster device). For an all-write workload the latency is determined by the slowest device. This slower device is either the offload device, if we always offload, or D_1. In all cases, offloading masks any switch-on costs.

5.5 Metadata

Table 2 shows the number of files for four families the authors of this paper are part of. This data is biased towards families with tech-savvy members. However, the point we make in this section is not that this data is representative of the population at large. We only confirm an observation made by Strauss et al. [26] that the amount of metadata involved is small in all cases and could easily reside in a data center today, and/or be fully cached on most consumer devices. We do this while showing that ZZFS's metadata structures are reasonably efficient.

We measured the amount of data with $R = 1$ and extrapolated for $R = 3$. The amount of metadata is calculated from ZZFS's metadata structures and is a function of the replication factor and number of files. It is interesting to observe that the second family has relatively fewer media files, and hence the average file size is much smaller than the other families. This translates to a higher relative metadata cost. Intuitively, the ratio of metadata to data decreases with larger file sizes.

Family	R	#files	data(GB)	metadata(MB)-%
1	1	23291	582	11 (0.0019%)
	3	23291	1746	68 (0.0038%)
2	1	3177	2.44	1.6 (0.06%)
	3	3177	7.32	9.3 (0.12%)
3	1	31621	705	15 (0.002%)
	3	31621	2116	93 (0.004%)
4	1	124645	164	61 (0.036%)
	3	124645	492	365 (0.07%)

Table 2: In ZZFS, the size of metadata is *O(numfiles x numdevices)*. This table shows the **total** data and metadata size for existing files of some of the authors. Files included are "Documents," "Pictures," "Videos" and "Music." *R* is the replication factor.

6 Related work

Data placement on devices and servers: AFS [8] and Coda [10] pioneered the use of a single namespace to manage a set of servers. AFS requires that client be connected with AFS servers, while Coda allows disconnected operations. Clients cache files that have been hoarded. BlueFS [16] allows for disconnected operation, handles a variety of modern devices and optimizes data placement with regard to energy as well. EnsemBlue [19] improved on BlueFS by allowing for a peer-to-peer dissemination of updates, rather than relying on a central file server. In Perspective, Salmon et al. use the *view* abstraction to help users set policies, based on metadata tags, about which files should be stored on which devices [23]. Recent work on Anzere [22] and PodBase [20] emphasizes the richness of the data placement policy space for home users.

An implicit assumption of the above work is that home users know how to set up these policies. This assumption might have been borrowed from enterprise systems, where data placement decisions can be automated and are guided by clear utility functions [27]. Our low-power communication channel can help with the execution of the above policies and can be used by most of the above systems as an orthogonal layer. It ensures that devices are awoken appropriately when the storage protocols need them to. While our design is compatible with the above work, ZZFS's choice of specific policies for data placement is arguably simpler than in the above work. It stems from our belief that, for many users, it takes too much time and effort to be organized enough to specify placement and replication policies like in Perspective or Anzere. ZZFS shows that in many common cases, no user involvement is required at all.

Consistency: Cimbiosys [21] and Perspective [23] allow for eventual consistency. Cimbiosys permits content-based partial replication among devices and is designed to support collaboration (e.g., shared calendars). Bayou [28] allows for application-specific conflict resolution. Our work can help the user's perception of consistency and reduces the number of accidental conflicts. In a system with eventual consistency, the low-power communication channel can be seen as helping reduce the "eventual" time to reach consistency, by turning dormant devices on appropriately.

File system best practices: ZZFS builds on considerable work on best-practices in file system design. For example, our distributed storage system has a NASD-based architecture [7], where metadata accesses are decoupled from data accesses and file naming is decoupled from location. The system is device-transparent [26]. The I/O director maintains versioned histories of files that can later be merged and is based on I/O offloading [15, 29].

User-centered design: We were inspired by a user-centered approach to system design. This was manifest not only in undertaking a small version of user research ourselves (Section 4), but by reference to the findings in the HCI literature in general. This literature still remains small on the topic dealt with here (e.g., see [20, 23] and also [5, 13, 17, 18]), but nevertheless helped provide some of the insights key to the technical work which is the main contribution of the paper.

7 Summary

Unpredictable networks and user behavior and non-uniform energy-saving policies are a fact of life. They act as barriers to the execution of well-intended personal storage system policies. This paper's contribution is to manage better these inherent uncertainties. We designed to enable a world in which devices can be rapidly turned on and off and are always network aware, even when off or dormant. The implications for the file system were illustrated through the implementation of ZZFS, a distributed device and cloud file system, designed for spontaneous and rather *ad hoc* file accesses.

8 Acknowledgments

We learned from the feedback of our shepherd Atul Adya, the anonymous reviewers, Tim Harris and Austin Donnelly, as well as several others at Microsoft.

References

[1] M. Abd-El-Malek, W. V. Courtright II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa Minor: versatile cluster-based storage. In *Proc. USENIX Conference on File and Storage Technologies (FAST)*, Dec. 2005.

[2] Y. Agarwal, S. Hodges, R. Chandra, J. Scott, P. Bahl, and R. Gupta. Somniloquy: augmenting network interfaces to reduce PC energy usage. In *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 365–380, Boston, Massachusetts, 2009.

[3] Y. Agarwal, S. Savage, and R. Gupta. Sleepserver: a software-only approach for reducing the energy consumption of PCs within enterprise environments. In *Proceedings of the 2010 USENIX annual technical conference*, Boston, MA, 2010. USENIX Association.

[4] Dropbox. Dropbox. `https://www.dropbox.com`.

[5] W. K. Edwards, M. W. Newman, and E. S. Poole. The infrastructure problem in HCI. In *CHI '10: Proceedings of the International Conference on Human factors in Computing Systems*, Atlanta, GA, 2010.

[6] W. W. Gaver. Technology affordances. In *CHI '91: Proceedings of the International Conference on Human factors in Computing Systems*, New Orleans, Louisiana, United States, 1991.

[7] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, CA, Oct. 1998.

[8] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6:51–81, February 1988.

[9] G. Hunt and D. Brubacher. Detours: Binary interception of Win32 functions. In *Proc. USENIX Windows NT Symposium*, Seattle, WA, July 1999.

[10] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda File System. *ACM Trans. Comput. Syst.*, 10(1):3–25, 1992.

[11] Lieberman software. White paper: Wake on LAN technology. `http://www.liebsoft.com/pdfs/Wake_On_LAN.pdf`.

[12] C. Marshall. Personal archiving 2011 keynote: People are people and things change. `http://research.microsoft.com/en-us/people/cathymar/pda2011-for-web.pdf`.

[13] M. L. Mazurek, J. P. Arsenault, J. Bresee, N. Gupta, I. Ion, C. Johns, D. Lee, Y. Liang, J. Olsen, B. Salmon, R. Shay, K. Vaniea, L. Bauer, L. F. Cranor, G. R. Ganger, and M. K. Reiter. Access control for home data sharing: Attitudes, needs and practices. In *CHI '10: Proceedings of the 28th International Conference on Human Factors in Computing Systems*, Atlanta, Georgia, USA, 2010.

[14] Microsoft. Windows Live Mesh. `http://explore.live.com/windows-live-mesh`.

[15] D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. Rowstron. Everest: Scaling down peak loads through I/O off-loading. In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, 2008.

[16] E. B. Nightingale and J. Flinn. Energy-efficiency and storage flexibility in the Blue File System. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 363–378, San Francisco, CA, 2004.

[17] W. Odom, A. Sellen, R. Harper, and E. Thereska. Lost in translation: Understanding the possession of digital things in the cloud. In *CHI '12: Proceedings of the International Conference on Human factors in Computing Systems*, Austin, TX, 2012.

[18] W. Odom, J. Zimmerman, and J. Forlizzi. Teenagers and their virtual possessions: Design opportunities and issues. In *CHI '11: Proceedings of the International Conference on Human factors in Computing Systems*, Vancouver, Canada, 2011.

[19] D. Peek and J. Flinn. EnsemBlue: integrating distributed storage and consumer electronics. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, Seattle, WA, 2006.

[20] A. Post, J. Navarro, P. Kuznetsov, and P. Druschel. Autonomous storage management for personal devices with PodBase. In *Proceedings of the 2011 USENIX annual technical conference*, Portland, OR, 2011. USENIX Association.

[21] V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, M. Walraed-Sullivan, T. Wobber, C. C. Marshall, and A. Vahdat. Cimbiosys: a platform for content-based partial replication. In *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 261–276, Boston, Massachusetts, 2009.

[22] O. Riva, Q. Yin, D. Juric, E. Ucan, and T. Roscoe. Policy expressivity in the Anzere personal cloud. In *2nd ACM Symposium on Cloud Computing (SOCC)*, Cascais, Portugal, 2011.

[23] B. Salmon, S. W. Schlosser, L. F. Cranor, and G. R. Ganger. Perspective: Semantic data management for the home. In *In Proc. USENIX Conference on File and Storage Technologies (FAST)*, San Francisco, CA, 2009.

[24] S. Sobti, N. Garg, F. Zheng, J. Lai, Y. Shao, C. Zhang, E. Ziskind, A. Krishnamurthy, and R. Y. Wang. Segank: A distributed mobile storage system. In *In Proc. USENIX Conference on File and Storage Technologies (FAST)*, pages 239–252, San Francisco, CA, 2004.

[25] J. Sorber, N. Banerjee, M. D. Corner, and S. Rollins. Turducken: hierarchical power management for mobile devices. In *Proceedings of the 3rd international conference on Mobile systems, applications, and services*, MobiSys '05, pages 261–274, Seattle, Washington, 2005. ACM.

[26] J. Strauss, J. M. Paluska, C. Lesniewski-Laas, B. Ford, R. Morris, and F. Kaashoek. Eyo: device-transparent personal storage. In *Proceedings of the 2011 USENIX annual technical conference*, Portland, OR, 2011. USENIX Association.

[27] J. D. Strunk, E. Thereska, C. Faloutsos, and G. R. Ganger. Using utility to provision storage systems. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, San Jose, California, 2008. USENIX Association.

[28] 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: Proceedings of the fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountain, Colorado, United States, 1995.

[29] E. Thereska, A. Donnelly, and D. Narayanan. Sierra: practical power-proportionality for data center storage. In *Proceedings of Eurosys'11*, pages 169–182, Salzburg, Austria, 2011. ACM.

[30] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger. Argon: performance insulation for shared storage servers. In *In Proc. USENIX Conference on File and Storage Technologies (FAST)*, San Jose, CA, 2007.

[31] Webdav.org. Webdav resources, 2010.

Revisiting Storage for Smartphones

Hyojun Kim [*], Nitin Agrawal, Cristian Ungureanu
NEC Laboratories America
hyojun.kim@cc.gatech.edu, nitin@nec-labs.com, cristian@nec-labs.com

Abstract

Conventional wisdom holds that storage is not a big contributor to application performance on mobile devices. Flash storage (the type most commonly used today) draws little power, and its performance is thought to exceed that of the network subsystem. In this paper we present evidence that storage performance does indeed affect the performance of several common applications such as web browsing, Maps, application install, email, and Facebook. For several Android smartphones, we find that just by varying the underlying flash storage, performance over WiFi can typically vary between 100% to 300% across applications; in one extreme scenario the variation jumped to over 2000%. We identify the reasons for the strong correlation between storage and application performance to be a combination of poor flash device performance, random I/O from application databases, and heavy-handed use of synchronous writes; based on our findings we implement and evaluate a set of pilot solutions to address the storage performance deficiencies in smartphones.

1 Introduction

Mobile phones, tablets, and ultra-portable laptops are no longer viewed as the wimpy siblings of the personal computer; for many users they have become the dominant computing device for a wide variety of applications. According to a recent Gartner report, within the next three years, mobile devices will surpass the PC as the most common web access device worldwide [38]. By 2013, over 40% of the enhanced phone installed-base will be equipped with advanced browsers [57].

Research pertaining to mobile devices can be broadly split into applications and services, device architecture, and operating systems. From a systems perspective, research has tackled many important aspects: understanding and improving energy management [36, 59, 26], network middleware [53], application execution models [30, 29], security and privacy [25, 32, 34, 39], and usability [27]. Prior research has also addressed several important issues centered around mobile functionality [55, 65], data management [66], and disconnected access [49, 37]. However, one important component is conspicuously missing from the mobile research landscape – storage performance.

[*]Work done as an intern, now at Georgia Institute of Technology

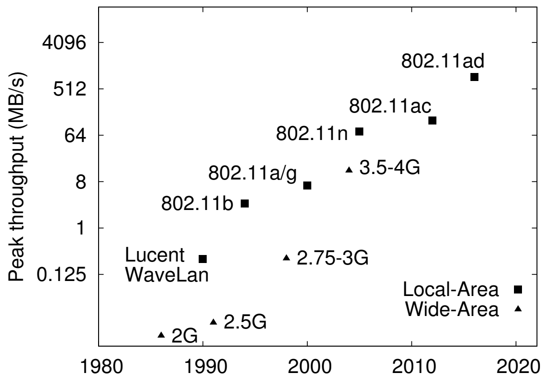


Figure 1: **Peak throughput of wireless networks.** Trends for local and wide-area wireless networks over past three decades; y-axis is log base 2.

Storage has traditionally not been viewed as a critical component of phones, tablets, and PDAs – at least in terms of the expected performance. Despite the impetus to provide faster mobile access to content locally [40] and through cloud services [61], performance of the underlying storage subsystem on mobile devices is not well understood. Our work started with a simple motivating question: does storage affect the performance of popular mobile applications? Conventional wisdom suggests the answer to be *no*, as long as storage performance exceeds that of the network subsystem. We find evidence to the contrary – even interactive applications like web browsing slow down with slower storage.

Storage performance on mobile devices is important for end-user experience today, and its impact is expected to grow due to several reasons. First, emerging wireless technologies such as 802.11n (600 Mbps peak throughput) [68] and 802.11ad (or "60 GHz", 7 Gbps peak throughput) offer the potential for significantly higher network throughput to mobile devices [41]. Figure 1 presents the trends for network performance over the last several decades; local-area networks are not necessarily the de-facto bottleneck on modern mobile devices. Second, while network throughput is increasing phenomenally, latency is not [62]. As a result, access to several cloud services benefits from a *split* of functionality between the cloud and the device [29], placing a greater burden on local resources including storage [51]. Third, mobile devices are increasingly being used as the primary computing device, running more performance intensive tasks than previously imagined. Smartphone usage is on the rise; smartphones and tablet computers are becoming a popular replacement for laptops [23]. In developing economies, a mobile/enhanced phone is often the only computing device available to a user for a variety of needs.

In this paper, we present a detailed analysis of the I/O behavior of mobile applications on Android-based smartphones and flash storage drives. We particularly focus on popular applications used by the majority of mobile users, such as, web browsing, app install, Google Maps, Facebook, and email. Not only are these activities available on almost all smartphones, but they are done frequently enough that performance problems with them negatively impacts user experience. Further, we provide pilot solutions to overcome existing limitations.

To perform our analysis, we build a measurement infrastructure for Android consisting of generic firmware changes and a custom Linux kernel modified to provide resource usage information. We also develop novel techniques to enable detailed, automated, and repeatable measurements on the internal and external smartphone flash storage, and with different network configurations that are otherwise not possible with the stock setup; for automated testing with GUI-based applications, we develop a benchmark harness using MonkeyRunner [16].

In our initial efforts, we propose and develop a set of pilot solutions that improve the performance of the storage subsystem and consequently mobile applications. Within the context of our Android environment, we investigate the benefits of employing a small amount of phase-change memory to store performance critical data, a RAID driver encompassing the internal flash and external SD card, using a log-structured file system for storing the SQLite databases, and changes to the SQLite `fsync` codepath. We find that changes to the storage subsystem can significantly improve user experience; our pilot solutions demonstrate possible benefits and serve as references for deployable solutions in the future.

As the popularity of Android-based devices surges, the setup we have examined reflects an increasingly relevant software and hardware stack used by hundreds of millions of users worldwide; understanding and improving the experience of mobile users is thus a relevant research thrust for the storage community. Through our analysis and design we make several observations:

Storage affects application performance: often in unanticipated ways, storage affects performance of applications that are traditionally thought of as CPU or network bound. For example, we found web browsing to be severely affected by the choice of the underlying storage; just by varying the underlying flash storage, performance of web browsing over WiFi varied by 187% and over a faster network (setup over USB) by 220%. In the case of a particularly poor flash device, the variation exceeded 2000% for WiFi and 2450% for USB.

Speed class considered irrelevant: our benchmarking reveals that the "speed class" marking on SD cards is not necessarily indicative of application performance; although the class rating is meant for sequential performance, we find several cases in which higher-grade SD cards performed worse than lower-grade ones overall.

Slower storage consumes more CPU: we observe higher total CPU consumption for the same application when using slower cards; the reason can be attributed to deficiencies in either the network subsystem, the storage subsystem, or both. Unless resolved, lower performing storage not only makes the application run slower, it also increases the energy consumption of the device.

Application knowledge ensues efficient solutions: leveraging a small amount of domain or application knowledge provides efficiency, such as in the case of our pilot solutions; hardware and software solutions can both benefit from a better understanding of how applications are using the underlying storage.

The contributions of this paper are threefold. First, we describe our measurement infrastructure that enables custom setup of the firmware and software stack on Android-devices to perform in-depth I/O analysis; along with the systems software, we contribute a set of benchmarks that automate several popular GUI-based applications. Second, we present a detailed analysis of storage performance on real Android smartphones and flash devices; to the best of our knowledge, no such study currently exists in the research literature. We find a strong correlation between storage and performance of common applications and contribute all our research findings. Third, we propose and evaluate pilot solutions to address the performance issues on mobile devices.

Based on our experimental findings and observations we believe improvements in the mobile storage stack can be made along multiple dimensions to keep up with the increasing demands placed on mobile devices. Storage device improvements alone can account for significant improvements to application performance. Device manufacturers are actively looking to bring faster devices to the mobile market; Samsung announced the launch of a PCM-based multi-chip package for mobile handsets [60]. Mobile I/O and memory bus technology needs to evolve as well to sustain higher throughput to the devices. Limitations in the systems software stack can however prevent applications from realizing the full potential of hardware improvements; we believe changes are also warranted in the mobile software stack to complement the hardware.

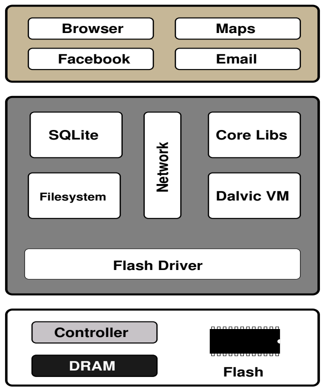


Figure 2: **Android Architecture.**

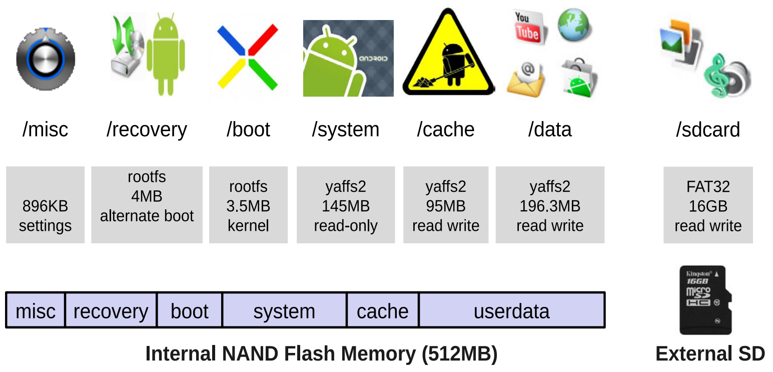


Figure 3: **Overview of Android's Storage Schema.**

Partition	Function	Size and Type
misc	Miscellaneous system settings (*e.g.*, Carrier ID, USB config, hardware settings, IMEI number); persistent shared space for OS and bootloader to communicate	896 KB
recovery	Alternative boot-into-recovery partition for advanced recovery and maintenance ops	4 MB, rootfs
boot	Enables the phone to boot, includes the bootloader and kernel/initrd	3.5 MB, rootfs
system	Contains remaining OS, pre-installed system apps, and user interface; typically read-only	145 MB, yaffs2
cache	Android can use it to stage and apply "over the air" updates; holds system images	95 MB, yaffs2
data	Stores user data (*e.g.*, contacts, messages, settings) and installed applications; SQLite database containing app data also stored here. Factory reset wipes this partition	196 MB, yaffs2
sdcard	External SD card partition to store media, documents, backup files etc	multi-GB, FAT32
sd-ext	Additional partition on SD card that can act as `data` partition, setup is possible through a custom ROM and `data2SD` software; non-standard Android partition	Varies

Table 1: **Data storage partitions for Android.** Partitions on internal flash and external SD card for Nexus One phone.

2 Mobile Device Overview

2.1 Android Overview

We present a brief overview of Android as it pertains to our storage analysis and development. Figure 2 shows a simplified Android stack consisting of flash storage, operating system and Java middleware, and applications; the OS itself is based on Linux and contains low-level drivers (*e.g.*, flash memory, network, and power management), Dalvik virtual machine for application isolation and memory management, several libraries (*e.g.*, SQLite, libc), and an application framework for development of new applications using system services and hardware.

The Dalvik VM is a fast register-based VM providing a small memory footprint; each application runs as its own process, with its own instance of the Dalvik VM. Android also supports "true" multitasking and several applications run as background processes; processes continue running in the background when user leaves an application (*e.g.*, a browser downloading web pages). Android's web browser is based on the open-source WebKit engine [4]; details on Android architecture and development can be found on the developer website [2].

2.2 Android Storage Subsystem

Most mobile devices are provisioned with an internal flash storage, an external SD card slot, and a limited amount of RAM. In addition, some devices (*e.g.*, LG G2X phone) also have a non-removable SD partition inside the phone; such storage is still treated as external.

Figure 3 shows the internal raw NAND and external flash storage on the Google Nexus One phone. The internal flash storage contains all the important system partitions, including partitions for the bootloader and kernel, recovery, system settings, pre-installed system applications, and user-installed application data. The external storage is primarily used for storing user content such as media files (*i.e.*, songs, movies, and photographs), documents, and backup images. Table 1 presents the functionality of the partitions in detail; this storage setup is fairly typical across Android devices.

Applications can store configuration and data on the device's internal storage as well as on the external SD card. Android uses SQLite [22] database as the primary means for storage of structured data. SQLite is a transactional database engine that is lightweight, occupying a small amount of disk storage and memory; it is thus popular on embedded and mobile operating systems. Applications are provided a well defined interface to create, query, and manage their databases; one or more SQLite databases are stored per application on `/data`.

The YAFFS2 [52] file system managing raw NAND flash was traditionally the file system of choice for the various internal partitions including `/system` and `/data`; it is lightweight and optimized for flash storage. Recently, Android transitioned to Ext4 as the default file system for these partitions [64]. Android provides a filesystem-like interface to access the external storage as well, with FAT32 as the commonly used file system on SD cards for compatibility reasons.

We believe the storage architecture described in this section is similar for other mobile operating systems as well; for example, Apple's iOS also uses SQLite to store application data. iOS Core Data is a data model framework built on top of SQLite; it provides applications access to common functionality such as save, restore, undo and redo. iOS 4 does not have a central file storage architecture, rather every file is stored within the context of an application. We focus on Android, since it allows systems-level development.

3 Android Measurement Setup

Since setting up smartphones for systems analysis and development is non-trivial, we describe our process here in detail; we believe this setup can be useful for someone conducting storage research on Android devices.

3.1 Mobile Device Setup

In this paper we present results for experiments on the Google Nexus One phone [12]. We also performed the same or a subset of experiments on the HTC Desire [13], LG G2X [15], and HTC EVO [14]; the results were similar and are omitted to save space.

The Nexus One is a GSM phone with a 1 GHz Qualcomm QSD8250 Snapdragon processor, 512 MB RAM, and 512 MB internal flash storage; the phone is running Android Gingerbread 2.3.4, the CyanogenMod 7.1.0 firmware [10] or the Android Open Source Project (AOSP) [3] distribution (as needed), and a Linux kernel 2.6.35.7 modified to provide resource usage information. We present a brief description of the generic OS customizations, which are fairly typical, and then explain the storage-specific customization later in this section.

In order to prepare the phones for our experiments, we setup the Android Debug Bridge (ADB) [1] on a Linux machine running Ubuntu 10.10. ADB is a command-line tool provided as part of Android developer platform tools that lets a host computer communicate with an Android device; the target device needs to be connected to the host via USB (in the USB debugging mode) or via TCP/IP. We subsequently *root* the device with unrevoked3 [20] to flash a custom recovery image (ClockworkMod [7]).

For our experiments we needed to bypass some of the constraints of the stock firmware; in particular, we needed support for *reverse tethering* the mobile device via USB, the ability to custom partition the storage, and access to a wider range of system tools and Linux utilities for development. For example, BusyBox [6] is a software application that provides many of the standard Linux tools within a single executable, ideal for an embedded device. CyanogenMod [10] is a custom firmware that provides these capabilities and is supported on a variety of smartphones. The Android Open Source Project (AOSP) [3] distribution provides capabilities similar to CyanogenMod but is supported only on a handful of Google-smartphones, including the Google Nexus One.

We used the CyanogenMod distribution for all experiments on non-Nexus phones, and for experiments that require comparison between a non-Nexus and the Nexus One phone (not shown in this paper). All Google Nexus One results presented in this paper exclusively use AOSP; we equipped both CyanogenMod and AOSP distributions with our measurement-centric customizations.

An important requirement, specific to our storage experiments, is to be able to compare and contrast application performance on different storage devices. Some of these applications heavily use the internal non-removable storage. In order to observe and measure all I/O activity, we change Android's `init` process to mount the different internal partitions on the external storage. Our approach is similar to the one taken by Data2SD [19]; in addition, we were able to also migrate to the SD card the `/system` and `/cache` partitions.

In order to adhere to Android's boot-time compatibility tests, we provided a 256 MB FAT32 partition at the beginning of the SD card, mounted as `/sdcard`. The `/system`, `/cache`, and `/data` partitions were formatted as Ext3; at the time we conducted our experiments, YAFFS2 and Ext3 were the pre-installed file systems on our test phones. We performed a preliminary comparison between Ext3 and Ext4 since Android announced the switch to Ext4 [64], but found the performance differences to be minor; a detailed comparison across several file systems can provide more useful data in the future.

Note that this setup is not normally used by end-users but allows us to run what-if scenarios with storage devices of different performance characteristics; the internal flash represents only a single data point in this set.

As part of our experiments, we want to understand the impact of storage on application performance under current WiFi networks, as well as under faster network connectivity (likely to be available in the future). For WiFi, we set up a dedicated wireless access point (IEEE 802.11 b/g) on a Dell laptop having 2GB RAM and an Intel Core2 processor. Since we do not have a faster wireless network on the phone, we emulate one by reverse tethering [21] it over the miniUSB cable connection with the same laptop

N/W	Rx	Tx
USB	8.04	7.14
WiFi	1.10	0.53

Table 2: **Network Performance.** Transfer rates for WiFi and USB reverse tether link with iperf (MB/s).

SD Card	Speed	Cost	Performance on desktop (MB/s)				Performance on phone (MB/s)			
(16 GB)	Class	US$	Sq W	Sq R	Rn W	Rn R	Sq W	Sq R	Rn W	Rn R
Transcend	2	26	4.16	18.03	1.18	2.57	4.35	13.52	1.38	2.92
RiData	2	27	**7.93**	16.29	**0.02**	2.15	5.86	11.51	0.03	2.76
Sandisk	4	23	5.48	12.94	0.68	1.06	4.93	8.44	0.67	0.73
Kingston	4	25	**4.92**	16.93	**0.01**	1.68	4.56	9.84	0.01	1.94
Wintec	6	25	15.05	16.34	0.01	3.15	9.91	13.38	0.01	3.82
A-Data	6	30	10.78	17.77	0.01	2.97	8.93	13.49	0.01	3.64
Patriot	10	29	10.54	17.67	0.01	2.96	8.83	13.38	0.01	3.72
PNY	10	29	15.31	17.90	0.01	3.56	10.28	14.02	0.01	3.95

Table 3: **Raw device performance and cost.** Measurements on Desktop with card reader (left) and on actual phone (right). "Sq" is sequential and "Rn" is random performance.

(allowing the device to access the internet connection of the host); Table 2 shows the measured performance of our WiFi and USB RT link using *iperf* [46].

To minimize variability due to network connections and dynamic content, we setup a local web server running Apache on the laptop. The webserver downloads the web pages that are to be visited during an experiment and caches them in memory; where available, we download the *mobile friendly* version of a web site.

We conducted all experiments on the internal non-removable flash storage and eight removable microSDHC cards, two each from the different SD speed classes [17]. Table 3 lists the SD cards along with their specifications and a baseline performance measurement done on a Transcend TS-RDP8K card reader[1] using the CrystalDiskMark benchmark V3.0.1 [9] (shown on the left side). The total amount of data written is 100 MB, random I/O size is 4KB, and we report average performance over 3 runs; observed standard deviation is low and we omit it from the table. Prices shown are as ordered from Amazon.com and its resellers, and Buy.com (to be treated as approximate). We also performed similar benchmarking experiments for the eight cards on the Nexus One phone itself, using our own benchmark program. Testing configuration is as before with 4KB random I/O size and 128 MB of sequential I/O; results in Table 3 (shown on the right side) exhibit a similar trend albeit lower performance than for desktop.

To summarize, read performance of the different cards is not a crucial differentiating factor and much better overall than the write performance. Sequential reads clearly show little or no correlation with the speed class; sequential write performance roughly improves with speed class, but with enough exceptions to not qualify as monotonic. Random read performance is not significantly different across the cards. The most surprising finding is for random writes: most if not all exhibit abysmal performance (0.02 MB/s or less!); even when sequential write performance quadruples (*e.g.*, Transcend versus Wintec), random writes perform several orders of magnitude worse.

[1] Note that internal flash could not be measured this way.

In terms of overall write performance including random and sequential, Kingston consistently performs the worst and tends to considerably skew the results; we try not to rely on Kingston results alone when making a claim about storage performance. In practice, we find that application performance varies even with the other better cards. Transcend performs the *best* for random writes, by as much as a factor of 100 compared to many cards, but performs the *worst* for sequential writes; Sandisk shows a similar trend. A-Data, Patriot, Wintec, and PNY perform poorly for random, but give very good sequential performance. Kingston and RiData suffer on both counts as they not only have poor random write performance, but also mediocre sequential write performance (shown in bold in Table 3); appliation-level measurements in §4 reflect the consequences of the poor microbenchmark results.

3.2 Measurement Software

We first explain our measurement environment and the changes introduced to collect performance statistics: (1) We made small changes to the microSD card driver to allow us to check "busyness" of the storage device by polling the status of the `/proc/storage_usage` file. (2) We wrote a background monitoring tool (*Monitor*) to periodically read the proc file system and store summary information to a log file; the log file is written to the internal `/cache` partition to avoid influencing the SD card performance. CPU, memory, storage, and network utilization information is obtained from `/proc/stat`, `/proc/meminfo`, `/proc/storage_usage` (busyness) and `/proc/diskstats`, and `/proc/net/dev` respectively. (3) We use `blktrace` [5] to collect block-level traces for device I/O.

In order to ascertain the overheads of our instrumentation, we conducted experiments with and without the measurement environment; we found that our changes introduce an overhead of less than 2% in total runtime.

Since many popular mobile applications are interactive, we needed a technique to execute these applications in a representative and reproducible manner; for this purpose we used the MonkeyRunner [16] tool to automate the execution of interactive applications. Our MonkeyRunner

App Name (Install)	Size (MB)	App Name (Launch)	Size (MB)
YouTube	1.95	AngryBird	18.65
Google Maps	6.65	SnowBoard	23.54
Facebook	2.96	Weather	2.60
Pandora	1.22	Imdb	1.38
Google Sky Map	2.16	Books	1.05
Angry Birds	18.65	Gallery	0.58
Music Download	0.70	Gmail	2.14
Angry Birds Rio	17.44	GasBuddy	1.88
Words With Friends	3.75	Twitter	1.36
Advanced Task Killer	0.10	YouTube	0.80

Table 4: **Apps for Install and Launch from Android Market.** Install: top Apps in Aug 2011, total size 55.58 MB, avg size 5.56 MB; Launch: 10 apps launched individually.

setup consists of a number of small programs put together to facilitate benchmarking with the necessary application; we illustrate the methodology next.

First, we start the Monitor tool to collect resource utilization information and note its PID. Second, we start the application under test using MonkeyRunner which defines "button actions" to emulate pressing of various keys on the device's touchscreen, for example, browsing forward and backward, zooming in and out with the touchscreen pinch, and clicking on screen to change display options. Third, while the various button actions are being performed, CPU usage is tracked in order to automatically determine the end of an interactive action. A class function `UntilIdle()` that we wrote is called from the MonkeyRunner script to detect the execution status of an app; it determines idle status using a specified *low CPU threshold* and the minimum time the app needs to stay below the threshold to qualify as idle. Fourth, once the sequence of actions is completed, we perform necessary cleanup actions and return to the default home screen. Fifth, the Monitor tool is stopped and the resource usage data is dumped to the host computer. Similar scripts are used to reset the phone to a known state in order to repeat the experiment (to compute mean and deviation).

3.3 Application Benchmarks

We now describe the Android apps that we use to assess the impact of storage on application performance; we automate a variety of popular and frequently used mobile apps to serve as benchmarks.

WebBench: is a custom benchmark program we wrote to measure web browsing performance in a non-interactive manner; it is based on the standard WebView Java Class provided by Android. WebBench visits a preconfigured set of web sites one after the other and reports the total elapsed time for loading the web pages. In order to accurately measure the completion time, we made use of the public method of WebView class named `onProgressChanged()`; when a web page is fully loaded, WebBench starts loading the next web page on the list. We ran WebBench to visit the top 50 web sites according to a recent ranking [8].

AppInstall: installs a set of top 10 Android apps on Google Android Market (listed in Table 4 on the left), successively, using the `adb install` command. App installation is an important and frequently performed activity on smartphones; each application on the phone once installed is typically updated several times during subsequent usage. In addition, often times a user needs to perform the install "on the go" based on location or situational requirements; for example, installing the IKEA app while shopping for furniture, or the GasBuddy app, when looking to refuel.

AppLaunch: launches a set of 10 Android apps using MonkeyRunner listed in Table 4 on the right; the apps are chosen to cover a variety of usage scenarios: games (AngryBird and SnowBoard) take relatively longer to load, read traffic to storage dominates. Weather and GasBuddy apps download and show real-time information from remote servers, *i.e.*, network traffic is high. Gmail and Twitter apps download and store data to local database, *i.e.*, both network and storage traffic is high. Books and gallery apps scan the local storage and display the list of contents, *i.e.*, read to storage dominates. Imdb has no storage or network traffic due to web cache hits, while YouTube launch is network intensive.

Facebook: uses the Facebook for Android application; each run constitutes the following steps: (a) sign into the author's Facebook account (b) load the news feed displayed initially on the phone screen (c) "drag" the screen five times to load more feed data (d) sign out.

Google Maps: uses the Google Maps for Android application; each run constitutes the following steps: (a) open the Maps application (b) enter origin and destination addresses, and get directions (c) zoom into the map nine times successively (d) switch from "map" mode to "satellite " mode (e) close application.

Email: uses the native email app in Android; each run constitutes the following steps: (a) open the app, (b) input account information, (c) wait until a list of received emails appears, and (d) close the application.

RLBench [56]: a synthetic benchmark app that generates a pre-defined number of various SQL queries to test SQLite performance on Android.

Pulse News [24]: a popular reader app that fetches news articles from a number of websites and stores them locally. Our benchmark consists of the following steps: (a) open Pulse app, (b) wait until news fetching process completes, and (c) close the app.

Background: another popular usage scenario is concurrent execution of two or more applications (Android and iOS are both multi-threaded); several apps run in the background to periodically "sync" data with a remote ser-

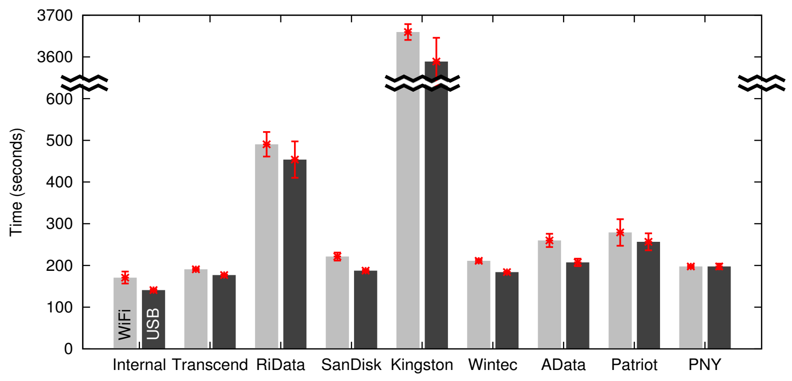


Figure 4: **Runtimes for WebBench on Google Nexus One.** Runtime for WebBench for SD cards and internal flash; each bar represents average over three trials with standard deviation; lighter bar is over WiFi, darker one for USB RT.

Activity	Write (MB)		Read (MB)	
	Sq	Rn	Sq	Rn
WebBench	41.3	32.2	6.8	0.5
AppInstall	123.1	5.6	0.7	0.1
Email	1.0	2.2	1.1	0.1
Maps	0.2	0.3	0	0
Facebook	2.0	3.1	0	0
RLBench	25.6	16.8	0	0
Pulse	2.6	1.0	0	0

Table 5: **I/O Activity Breakdown.** Aggregate seq. and random, writes and reads during benchmark; note moderate to high rand:seq write ratios for WebBench, Email, Maps, Facebook, and low for AppInstall. Zero value means no activity during run.

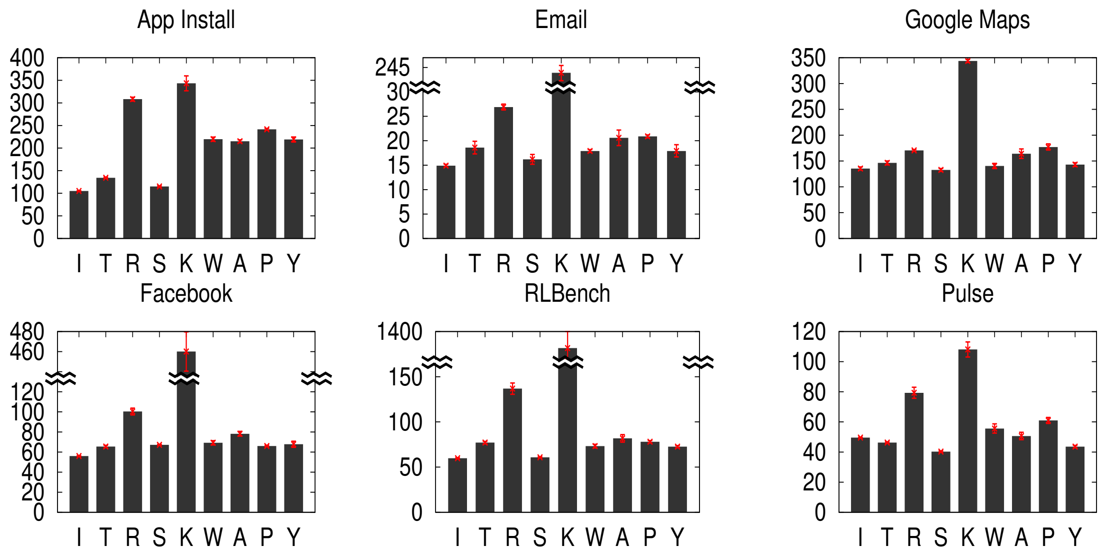


Figure 5: **Runtimes for popular applications.** Similar to Fig 4 but for several other apps on WiFi only; I: Internal, T: Transcend, R: RiData, S: Sandisk, K: Kingston, W: Wintec, A: AData, P: Patriot, Y: PNY. Some graphs are plotted with a discontinuous y-axis to preserve clarity of the figure in presence of outliers like Kingston.

vice or to provide proactive notifications. Our benchmark consists of the following set of apps in auto sync mode: Twitter, books, contacts, Gmail, Picasa, and calendar, and a set of active widgets: Pulse, news, weather, YouTube, calendar, Facebook, Market, and Twitter.

For many of the above benchmarks (*e.g.*, Facebook, Email, Pulse, Background), the actual contents and amount of data can vary across runs; we measure the total amount of data transferred and normalize the results per Megabyte. We also repeat the experiment several times to measure variations; for multiple iterations, the local application cache is deleted following each run.

4 Performance Evaluation

In this section we present detailed measurement results for application runtime performance, application launch times, concurrent app execution, and CPU consumption.

4.1 Application Runtime Performance

The first set of experiments compare the performance of WebBench on internal flash and the eight SD cards described earlier. Figure 4 shows the runtime of WebBench for WiFi and USB reverse tethering.

Surprisingly, even with WiFi, we notice a 187% performance difference between the internal flash and RiData; for Kingston, the difference was a whooping 2040%. To ensure that the Kingston results were not due to a defective device, we repeated the experiments with two more new Kingston cards from two different speed classes; we found the results to be similarly poor. Here onwards, so as to not rely on Kingston alone when making a claim about application performance, we mention the difference both with the second-worst and worst performing card for any given experiment.

As expected, the faster the network (USB RT) the

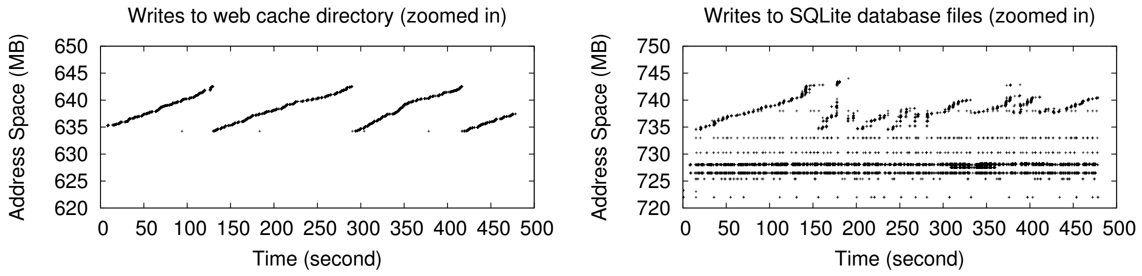


Figure 6: **SQLite I/O pattern.** The left graph shows write I/O to the webcache directory contents on /data, on right are writes to SQLite database files; reads are comparatively less and omitted from presentation.

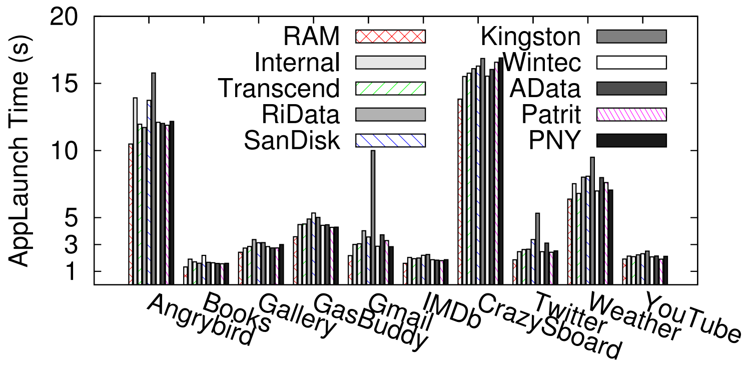


Figure 7: **Application Launch.** Launch times (secs) for several popular apps on 8 SD cards, internal flash, and a memory-backed RAMdisk.

App	R	W	Rx	Tx
AngryBird	20.69	0.04	4.09	4.44
SnowBoard	20.92	0.02	1.87	0.53
Weather	8.72	0.07	16.11	2.56
Imdb	2.71	0.00	0.08	0.00
Books	2.98	0.00	0.00	0.00
Gallery	1.88	0.00	0.00	0.00
Gmail	3.20	0.05	3.00	0.93
GasBuddy	7.47	0.00	2.28	0.80
Twitter	4.62	0.06	5.63	1.61
YouTube	2.06	0.00	65.47	4.83

Table 6: **App Launch Summary.** Total data (MB) read and written to storage and transferred over the network for the set of apps launched.

higher the impact of storage: 222% difference between internal and RiData, 2450% for Kingston. We find a similar trend for several popular apps; Figure 5 shows the results over WiFi for AppInstall, email, Google Maps, Facebook, RLBench, and Pulse. Since the phenomenon of storage and application performance correlation is clearly identifiable with existing WiFi networks, we hereafter omit results for the USB network. The difference between the best and worst case performance varies from 195% (225%) for AppInstall, 80% (1670%) for email, 60% (660%) for Maps, 80% (575%) for Facebook, 130% (2210%) for RLBench, and 97% (168%) for Pulse; Kingston numbers are shown in parentheses.

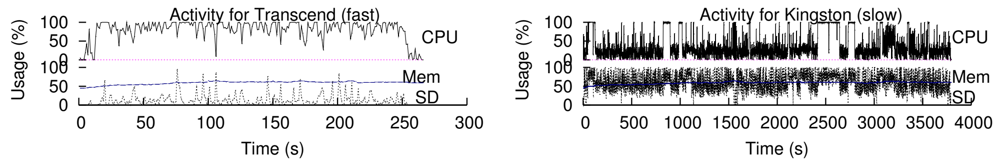
To better understand why storage affects application performance, we present in Table 5 presents a breakdown of the I/O activity during various workload runs. Amount of reads is less than writes for all workloads. In the case of WebBench roughly 1.3 times more data is written sequentially than randomly. Since the difference between sequential and random performance is at least a factor of 3 for all SD cards (see Table 3), the time to complete the random writes dominates; the same holds true for the other applications in the table. Although not shown in the table, the `/data` partition receives most of the I/O, with only a few reads going to the `/system` partition.

The disparity between sequential and random write performance is inherent with flash-based storage; our evaluation results suggest this to be one of the primary reasons behind the slower performance. However, this still doesn't explain the presence of the random writes and overwrites even for seemingly sequential application needs. In order to understand this we take a closer look at the applications and their usage of Android storage.

The storage schema used by the browser application consists of the *cache* as the unstructured web cache storing image and media files and two SQLite database files; *webview.db* is a database for application settings and preferences and *webviewCache.db* stores an index to manage the web cache. The database files are much smaller in size compared to the cache; in our setup, the cache consisted of 315 files totaling 6MB whereas the database files were 34KB and 137KB for webview.db and webviewCache.db respectively. Figure 6 shows the write pattern to the web cache directory and the SQLite database files; web cache writes are mostly sequential with reuse of the same address space over time; SQLite exhibits a high degree of random writes and updates to the same block addresses. Since by default the database writes are synchronous, each write causes a (often unnecessary) delay.

4.2 Application Launch

Application launch is an important performance metric [47], especially for mobile users. Figure 7 shows the time taken to launch a number of Android applications on the various flash storage devices; Table 6 lists those apps along with a summary of disk I/O reads and writes, and



Card	Fast	Slow
Idle	1.5	0.7
ioWait	1.6	232.7
Active	20.8	133.3
Total	23.9	366.7

Figure 9: **Storage and CPU activity for WebBench on *fast* and *slow* SD cards.** The graph on the left shows instantaneous CPU utilization, memory consumption, and storage busyness during the course of a WebBench run on the fast Transcend card; the graph on the right repeats the *same* experiment for the slow Kingston, taking considerably longer to finish. Table summarizes the aggregate CPU ticks (in thousands) used for WebBench; compare the active counts for fast and slow.

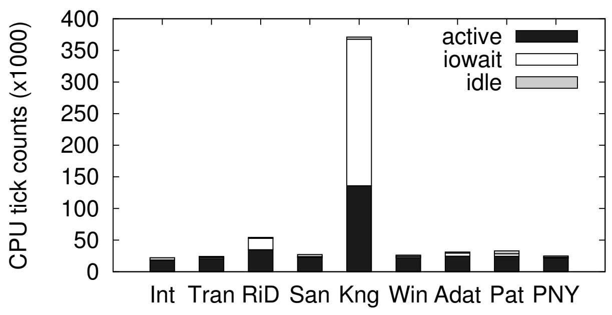


Figure 8: **Aggregate CPU for WebBench.** Stacked bar shows active, idle, and ioWait times on Nexus One; ioWait correlates with runtimes (Fig 4). Even active times vary across devices showing that some devices burn more CPU for same work!

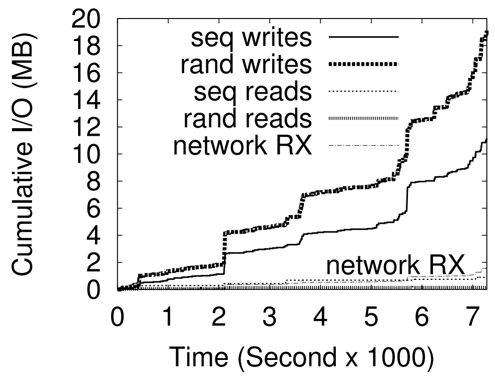


Figure 10: **Background I/O pattern.** Breakdown of I/O issued by Background apps in 2 hours.

data transferred over the network during the launch. Most apps take a few seconds to launch, with games taking upwards of 10 seconds. Larger apps (*e.g.*, games) tend to take a noticeable amount of time to launch, contrary to the target of "significantly less than 1 second to launch a new app" [31]. As seen in Figure 7, barring a few exceptions, the launch time varies between about 10% (for the Snowboard game) to 40% (for the Weather app); Twitter (120%) and Gmail (250%) showed the most variation.

In order to ascertain the upper bound of launch time improvement through storage, we placed *all* application data on a RAMdisk; the test is conducted with the PNY card storing the `/system`, `/sdcard`, `/cache` partitions and the `/data` partition mounted with `tmpfs`. To remove the effects of reading from `/system` and `/sdcard`, we warm the buffer cache; we verify the same by tracking all I/O to the flash storage. Launch times do not significantly change even when all data is being read from memory. Storage is likely not a significant contributor to app launch performance; research to speed up launch will perhaps benefit by focusing on other sources of delay such as application think time.

4.3 Concurrent Applications

Figure 10 shows I/O activity for a 7200 second run of the Background workload; during the period, the phone received about 1.6 MB of data over the network. Interestingly, the amount of data written to storage in the same period is 30 MB (a factor of roughly 20); the majority of writes are for updating application-specific data and indices to the SQLite databases. Although the storage throughput requirement is quite low, the additional random writes can cause latency spikes for foreground applications (not shown). With the Android development team's desire to minimize application switch time and provide the appearance of "all applications running all of the time" [31] (see section: "When does an application 'stop'?") for mobile devices, handling concurrent applications and their I/O demands can be an increasingly important challenge in the future.

4.4 CPU Consumption

Figure 8 shows the breakdown of CPU utilization for WebBench; the stacked bar chart shows the CPU tick counts during active, idle, and ioWait periods (a "tick" corresponds to 10ms on our phone); Figure 9 shows the CPU utilization and I/O busyness for the same experiment for two SD cards: a fast Transcend, and a slow Kingston. Since the non-idle, non-ioWait CPU consumption includes not only the contribution of the benchmark but also all background activities, we also measured CPU consumption for background activities alone (to subtract from the total). Note that this is unlike the set of background activities discussed in Section §3.3 as we turned off automatic syncing and active widgets; we find that the share of CPU consumption due to background tasks is less than 1% of the total.

The graphs reveal the interesting phenomenon that ag-

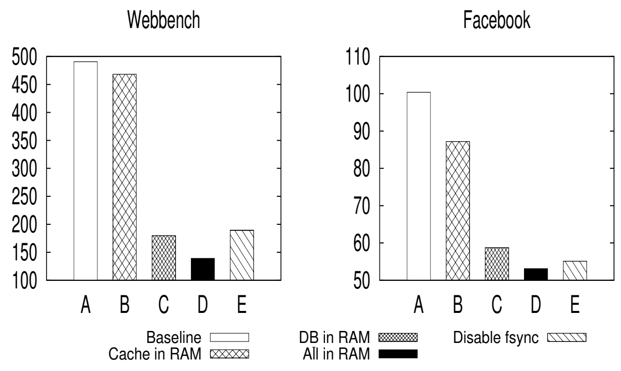


Figure 11: **What-If Performance Analysis.** Experiments were conducted for WebBench (left) and Facebook (right); data stored in memory using a RAMdisk and RiData card as the flash backing store where needed (*e.g.*, for baseline). Y-axis is Time in seconds; Solutions **A:** Baseline, **B:** Cache in RAM, **C:** DB in RAM, **D:** All in RAM, **E:** Disable Sync.

gregate CPU consumed for the same benchmark increases with a slower storage device (by just looking at the "active" component). This points to the fact that storage has an indirect impact on energy consumption by burning more CPU. Ideally, one would expect a fixed amount of CPU to be consumed for the same amount of work; since the results show CPU consumption to be disproportional to the amount of work, we hypothesize it being due to deficiencies in either the network subsystem, the storage subsystem, or both. We need to investigate this matter further to identify the root causes.

Slower storage also increases energy consumption in other indirect ways; for example, keeping the LCD screen turned on longer while performing interactive tasks, keeping the WiFi radio busy longer, and preventing the phone from going to a low-power mode sooner.

5 Pilot Solutions

We present potential improvements in application performance through storage system modifications. We start with a what-if analysis to provide the envelope of performance gains and then present a set of pilot solutions.

5.1 What-If Analysis

The detailed analysis of storage performance gave insights into the performance problems faced by applications, but before proposing actual solutions we wanted to understand the scope for potential improvements. We performed a set of *what-if* analyses to obtain the upper bounds on performance gains that could be achieved, for example, by storing all data in memory. For comparison sake, we performed experiments with both memory as the backing store (using RAMdisk) and SD cards as the backing store; in the different analysis experiments we placed different kinds of data on the RAMdisk, for example, the cache, or the database files. Figure 11 compares the relative benefits of the various approaches, as measured for the WebBench and Facebook workloads for the RiData card and a RAMdisk; the trends for the other SD cards were similar, although the actual gains were of course different with every card.

Placing the entire "cache" folder on RAM (bars B) does improve performance, but not by much (*i.e.*, 5% for WebBench and 15% for Facebook). Placing the SQLite database on RAM (bars C) however improves performance by factors of three and two for WebBench and Facebook respectively; placing both the cache and the database on RAM (bars D) does not provide significant additional benefit. Transforming the cache and database writes to be asynchronous (bars E) recoups most of the performance and performs comparably to the SQLite on RAM solution.

The performance evaluation in the previous section and the what-if analysis lead to the following conclusions: First, the key bottleneck is the "wimpy" storage prevalent today on mobile devices; even while the internal flash and the SD cards are increasingly being used for desktop like-workloads, their performance is significantly worse than storage media on laptops and desktops. Second, the Android OS exacerbates the poor storage performance through its choice of interfaces; the synchronous SQLite interface primarily geared for ease of application development is being used by applications that are perhaps better off with more light-weight consistency solutions. Third, the SQLite write traffic itself is quite random with plenty of synchronous overwrites to the flash storage causing further slowdown. Finally, apps use the Android interfaces oblivious to performance. A particularly striking example is the heavy-handed management of application caches through SQLite; the web browser writes a cache map to SQLite significantly slowing down the cache writes.

We implement and evaluate a set of pilot solutions to show the potential for improving user experience through improvements in the Android storage system; while not rigorous enough to serve as deployable solutions, these can evolve into robust and detailed solutions in the future. We classify the solution space into four categories:

- Better storage media for mobile devices to provide baseline improvements
- Firmware and device drivers to effectively utilize existing and upcoming storage devices
- Enhancements to mobile OS to avoid the storage bottlenecks and provide new functionality
- Application-level changes to judiciously use the supplied storage interfaces

Figure 12 shows the improvements through the pilot solutions for WebBench and Facebook using Kingston and RiData; as with the what-if analysis, trends for other SD

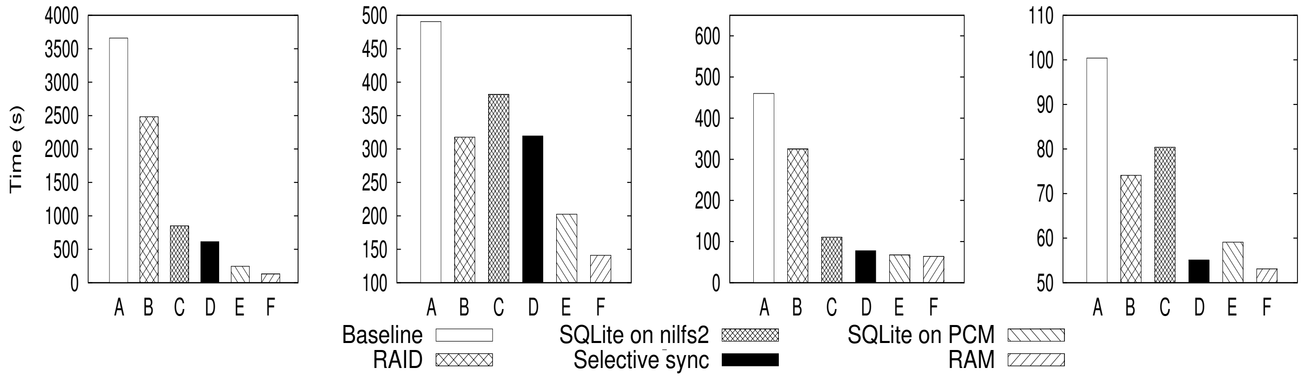


Figure 12: **Pilot Solutions.** Runtime results for WebBench (leftmost two) and Facebook (rightmost two) for the Kingston and RiData cards; y-axis is Time in seconds. Solutions **A:** Baseline, **B:** RAID over SD, **C:** SQLite on Nilfs2, **D:** Selective Sync, **E:** SQLite on PCM, **F:** All in RAM.

cards were similar but actual gains varied. Bars A in Figure 12 represent the baseline performance, while bars F are meant to represent an upper bound on performance with all data stored in RAM.

5.2 Storage Devices Not Wimpy Anymore

An obvious solution is to improve the performance of the storage device, *i.e.*, using better flash storage or a faster non-volatile memory such as PCM. Indeed, flash fabrication technology itself is improving at a fair pace; scaling trends project flash to double in capacity every two years until the year 2016 [45]. However, when it comes to performance, cost pressures in the consumer market are driving manufacturers to move away from the more reliable, higher performing SLC flash to the less reliable, lower performing MLC or TLC flash; this makes it harder to rely solely on improvements due to flash scaling. Our findings reveal that performance of a relatively small fraction of I/O traffic is responsible for a large fraction of overall application performance. A more efficient solution is thus to use the faster storage media as a persistent write buffer for the performance-sensitive I/O traffic: a small amount of PCM to buffer writes issued by the SQLite database can improve the performance.

We built a simple PCM emulator for Android to evaluate our solution; the emulator is implemented as a pseudo block-device based on the timing specifications from recent work [28], using memory as the backing store. The PCM buffer can be used as staging area for all writes or as the final location for the SQLite databases; our emulator can be configured with a small number of device-specific parameters. Figure 12 (bars E) show the performance improvements by using a small amount (16 MB) of PCM; in this experiment, PCM is used as the final location for only the database files.

An alternative approach, as envisioned by Pocket Cloudlets [51], is to rely on substantial augmentation of existing flash storage capabilities on mobile devices

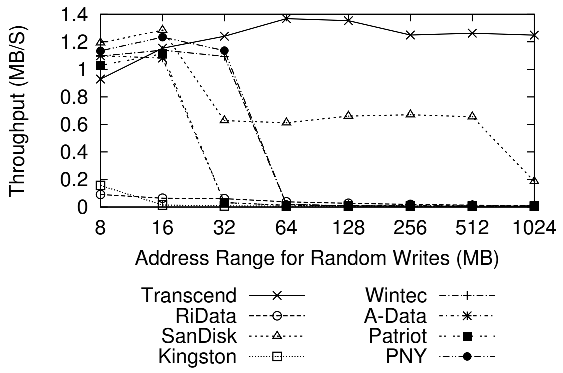


Figure 13: **Explanation of RAID Speedup.** Variation in throughput for SD cards with increasing write address range.

and/or full replacement of flash with PCM or STT-MRAM [43]. In reality, storage-class memory may be placed in different forms on the mobile system, for example, on the CPU-memory bus, or as backing store for the virtual memory. Our intent here was two-fold (a) understand the approximate benefits of using such a persistent buffer, and (b) demonstrate that even with a relatively small amount of PCM, significant gains can be made by judiciously storing performance-critical data; a deployed solution can certainly incorporate PCM in the storage hierarchy in better ways.

5.3 RAID over SD

Another solution is to leverage the I/O parallelism already existent on most phones: an internal flash drive and an external SD card. We built a simple software RAID driver for Android with I/O striped to the two devices (RAID-0) in 4 KB blocks. Note that a deployable solution will require more effort: (a) it would need to handle storage devices of potentially differing speeds (b) handle accidental removal of the external SD card.

While for some SD cards we obtained the expected improvements as in Figure 12 (bars B), *i.e.*, greater than 1X and less than 2X, for others we obtained a speedup greater than 2X (not shown); we suspected that this could be due to the idiosyncrasies of the FTL on the card. As many consumer flash devices employ the log-block wear-leveling scheme [48], their performance is sensitive to the write footprint; a reduction in the amount of random writes reduces the overhead of the garbage collection, improving the performance.

To verify our hypothesis, we performed another experiment. Figure 13 shows the throughput obtained for an increasing address range with random writes; the I/O size is 4KB and number of requests is 2048, totaling 8 MB of writes. In order to minimize the effects of FTL state being carried forward from the previous experiment, we sequentially write 1 GB of data before every run.

For Kingston, Wintec, A-Data, Patriot, and PNY, as the address range increases, the throughput drops significantly and then stabilizes at the low level; for RiData, throughput drops but not as sharply, while for Transcend the throughput remains consistently high (we do not have an explanation for the slight increase, multiple measurements gave similar results). Sandisk exhibits more than one regime change, dropping first around the 32 MB mark and then around the 1024 MB mark.

To explain our surprising performance improvements, in a log-block FTL, a small number of physical blocks are available for use as *log blocks* to stage an updated block; a one-to-one correspondence exists between logical and physical blocks. Since the amount of data written to one disk in a 2-disk RAID-0 array is roughly half of the total, the disk write footprint reduces and block address range shrinks; the RAID scheme simply pushes the operating regime of an SD card towards the left, and depending on the actual footprint, provides super-linear speedup! While we came across this performance variation in course of our RAID experiments, the implications are more generic; one can design other solutions centered around the compaction of the write address range.

5.4 Using a Log-structured File System

Log-structured file systems provide good performance for random writes [58]; another solution to alleviate the effects of the random writes is thus to place the database files on a log-structured file system. We used the Nilfs2 [50] file system on Android since it works with block devices; we created a separate partition on the phone's flash storage to store the entire SQLite database. Figure 12 (bars C) show the benefits of log-structuring; SQLite on Nilfs2 improves the performance of WebBench and Facebook by more than a factor of 4 for Kingston, and over 20% for RiData.

5.5 Application Modifications

Finally, several solutions are possible if one is able to modify either the SQLite interface or the applications themselves. We demonstrate the benefits of such techniques with a simple modification to SQLite: providing the capability to perform *selective* `sync` operations based on application-specific requirements; in our current implementation, we simply turn off sync for the database files that are deemed asynchronous as per our analysis (for example, the WebView database file serving as the index for the web cache). Figure 12 (bars D) compare the benefits of the selective sync operation with other previously proposed solutions, providing noteworthy benefits especially for Facebook.

Another potential technique to improve performance at the application level is through the use of larger transactions, amortizing the overhead of the SQLite sync interface. A careful restructuring of the application programming interface can perhaps lead to significant gains for future apps, but is beyond the scope for this paper; the interface discussion is a classic chicken-and-egg problem in the context of storage systems [54, 63]. Recently a new backend for SQLite has been proposed that uses write-ahead logging [18]; such techniques have the potential to ameliorate the random write bottleneck without requiring changes to the API.

5.6 Summary of Solutions

Through our investigation of the solution space we notice several avenues for further performance improvements in the storage subsystem on mobile devices, and consequently the end-user experience. Our analysis reveals that a small amount of domain or application knowledge can improve performance in a more efficient way; through our pilot solutions we demonstrate the potential benefits of explicit and implicit storage improvements.

Programmers tend to heavily use the general-purpose "all-synchronous" SQLite interface for its ease of use but end up suffering from performance shortcomings. We posit that a *data-oriented* I/O interface would be one that enables the programmer to specify the I/O requirements in terms of its reliability, consistency, and the property of the data, *i.e.*, temporary, permanent, or cache data, without worrying about how its stored underneath. For example, a key-value store specifically for cache data does not need to provide ultra-reliability; a web browser can use the cache key-value store as its web cache in a more performance-efficient manner than SQLite.

6 Related Work

We found little published literature on storage performance for mobile devices. One of the earliest works on storage for mobile computers [33] compares the performance of hard disks and flash storage on an HP OmniBook; remarkably, many of their general observations are still valid. Datalight [11], provider of data management technologies for mobile and embedded devices to OEMs, make an observation similar to ours with reference to their proprietary Reliance Nitro file system. According to their website, lack of device performance and responsiveness is one of the important shortcomings of the [Windows] Mobile platforms; OEMs using an optimized software stack can improve performance. Our results also reaffirm some of the recent findings for desktop applications on the Mac OS X [42]: lack of pure sequential access for seemingly sequential application requests, heavy-handed use of synchronization primitives, and the influence of underlying libraries on application I/O.

A recent study of web browsers on smartphones [67] examined the reasons behind slow web browsing performance and found that optimizations centering around compute-intensive operations provide only marginal improvements; instead "resource loading" (*e.g.*, files of various types being fetched from the webserver) contributes most to browser delay. While this work focuses more specifically on the browser and the network, it reaffirms the observation that improvements in the OS and hardware are needed to improve application performance.

Other related work has focused on the implications of network performance on smartphone applications [44] and on the diversity of smartphone usage [35]. Finally, there is extensive work in developing smarter, richer, and more powerful applications for mobile devices, far too much to cite here. We believe the needs of these applications are in turn going to drive the performance requirements expected of hardware devices, including storage, as well as the operating system software.

7 Conclusions

Contrary to conventional wisdom, we find evidence that storage is a significant contributor to application performance on mobile devices; our experiments provide insight into the Android storage stack and reveal its correlation with application performance. Surprisingly, we find that even for an interactive application such as web browsing, storage can affect the performance in non-trivial ways; for I/O intensive applications, the effects can get much more pronounced. With the advent of faster networks and I/O interconnects on the one hand, and a more diverse, powerful set of mobile apps on the other, the performance required from storage is going to increase in the future. We believe the storage system on mobile devices needs a fresh look and we have taken the first steps in this direction.

8 Acknowledgements

We thank the anonymous FAST reviewers and our shepherd, Raju Rangaswami, for their valuable feedback that improved the presentation of this paper. We thank Akshat Aranya for his assistance in setting up the Android test environment and experimental data analysis. We thank Kishore Ramachandran for providing several useful discussions and detailed comments on the paper.

References

[1] Android Debug Bridge (ADB). http://developer.android.com/guide/developing/tools/adb.html.

[2] Android Developers Website. http://developer.android.com/index.html.

[3] Android Open Source Project. http://source.android.com/index.html.

[4] Android WebKit Package. http://developer.android.com/reference/android/webkit/package-summary.html.

[5] Block I/O Layer Tracing: blktrace. http://linux.die.net/man/8/blktrace.

[6] Busybox unix utilities. http://www.busybox.net/about.html.

[7] Clockworkmod rom manager and recovery image. http://www.koushikdutta.com/2010/02/clockwork-recovery-image.html.

[8] Compete ranking of top 50 web sites for february 2011 reveals familiar dip. http://tinyurl.com/3ubxzbl.

[9] CrystalDiskMark Benchmark V3.0.1. http://crystalmark.info/software/CrystalDiskMark/index-e.html.

[10] Cyanogenmod. http://wiki.cyanogenmod.com/index.php?title=What_is_CyanogenMod.

[11] Datalight: Software for risk-free mobile data. http://www.datalight.com/solutions/linux-flash-file-system/performance-hardware-managed-media.

[12] Google nexus one. http://en.wikipedia.org/wiki/Nexus_One.

[13] Htc desire. http://www.htc.com/www/product/desire/specification.html.

[14] HTC EVO Phone . http://www.htc.com/us/products/evo-sprint#tech-specs. Retrieved in Sep 2011.

[15] LG G2X P999 Phone. http://www.lg.com/us/products/documents/LG-G2x-Datasheet.pdf. Retrieved in Sep 2011.

[16] MonkeyRunner for Android Developers. http://developer.android.com/guide/developing/tools/monkeyrunner_concepts.html.

[17] SD Speed Class/UHS Speed Class. https://www.sdcard.org/consumers/speed_class/.

[18] SQLite Backend with Write-Ahead Logging. http://www.sqlite.org/draft/releaselog/3_7_0.html.

[19] Starburst data2sd. http://starburst.droidzone.in/.

[20] Unrevoked 3: Set your phone free. http://unrevoked.com/recovery/.

[21] Usb reverse tethering setup for android 2.2. http://blog.mycila.com/2010/06/reverse-usb-tethering-with-android-22.html.

[22] Using databases in android: Sqlite. http://developer.android.com/guide/topics/data/data-storage.html#db.

[23] Motorola Webtop: Release Your Smartphone's True Potential. http://www.motorola.com/Consumers/US-EN/Consumer-Product-and-Services/WEBTOP/Meet-WEBTOP, 2011.

[24] Alphonso Labs. Pulse News Reader. https://market.android.com/details?id=com.alphonso.pulse&hl=en.

[25] J. Bickford, H. A. Lagar-Cavilla, A. Varshavsky, V. Ganapathy, and L. Iftode. Security versus energy tradeoffs in host-based mobile malware detection. In *MobiSys'11: Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, page TBD, Bethesda, Maryland, USA, June/July 2011. ACM Press, New York, NY, USA.

[26] A. Carroll and G. Heiser. An analysis of power consumption in a smartphone. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIX ATC'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[27] S. J. Castellucci and I. S. MacKenzie. Gathering text entry metrics on android devices. In *Proceedings of the 2011 Conference on Human Factors in Computing Systems (CHI)*, CHI EA '11, pages 1507–1512, New York, NY, USA, 2011. ACM.

[28] S. Chen, P. B. Gibbons, and S. Nath. Rethinking database algorithms for phase change memory. In *CIDR '11*, pages 21–31, Asilomar, CA, 2011.

[29] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 301–314, New York, NY, USA, 2011. ACM.

[30] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, MobiSys '10, pages 49–62, New York, NY, USA, 2010. ACM.

[31] Dianne Hackborn. Multitasking the Android Way. `http://android-developers.blogspot.com/2010/04/multitasking-android-way.html`, april 2010.

[32] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *20th USENIX Security Symposium*, San Francisco, CA, Aug. 2011.

[33] F. Douglis, R. Cáceres, M. F. Kaashoek, K. Li, B. Marsh, and J. A. Tauber. Storage alternatives for mobile computers. In *OSDI*, pages 25–37, 1994.

[34] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[35] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin. Diversity in smartphone usage. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, MobiSys '10, pages 179–194, New York, NY, USA, 2010. ACM.

[36] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, SOSP '99, pages 48–63, New York, NY, USA, 1999. ACM.

[37] J. Flinn, S. Sinnamohideen, N. Tolia, and M. Satyanarayanan. Data Staging on Untrusted Surrogates.

[38] Gartner. Gartner highlights key predictions for it organizations and users in 2010 and beyond. `http://www.gartner.com/it/page.jsp?id=1278413`.

[39] R. Geambasu, J. P. John, S. D. Gribble, T. Kohno, and H. M. Levy. Keypad: an auditing file system for theft-prone devices. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 1–16, New York, NY, USA, 2011. ACM.

[40] V. Gundotra and H. Barra. Android: Momentum, Mobile and More at Google I/O. Keynote at Google I/O, May 2011.

[41] D. Halperin, S. Kandula, J. Padhye, P. Bahl, and D. Wetherall. Augmenting data center networks with multi-gigabit wireless links. In *Proceedings of the ACM SIGCOMM 2011 conference*, SIGCOMM '11, pages 38–49, New York, NY, USA, 2011. ACM.

[42] 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 '11*, Cascais, Portugal, October 2011.

[43] Y. Huai. Spin-transfer torque MRAM (STT-MRAM): Challenges and Prospects. *AAPPS Bulletin*, 18(6):33–40, Dec. 2008.

[44] J. Huang, Q. Xu, B. Tiwana, Z. M. Mao, M. Zhang, and P. Bahl. Anatomizing application performance differences on smartphones. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, MobiSys '10, pages 165–178, New York, NY, USA, 2010. ACM.

[45] I. T. R. for Semiconductors Working Group. International technology roadmap for semiconductors. Technical report, International Technology Roadmap for Semiconductors, 2009.

[46] iperf network performance tool. http://sourceforge.net/projects/iperf.

[47] Y. Joo, J. Ryu, S. Park, and K. G. Shin. Fast: quick application launch on solid-state drives. In *Proceedings of the 9th USENIX conference on File and Storage Technologies*, FAST '11, 2011.

[48] J. Kim, J. M. Kim, S. 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.

[49] J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Trans. Comput. Syst.*, 10(1), February 1992.

[50] R. Konishi, Y. Amagai, K. Sato, H. Hifumi, S. Kihara, and S. Moriai. The linux implementation of a log-structured file system. *SIGOPS Oper. Syst. Rev.*, 40(3):102–107, 2006.

[51] E. Koukoumidis, D. Lymberopoulos, K. Strauss, J. Liu, and D. Burger. Pocket cloudlets. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 171–184, New York, NY, USA, 2011. ACM.

[52] C. Manning. YAFFS: Yet Another Flash File System. `http://www.aleph1.co.uk/yaffs`, 2004.

[53] P. Meroni, E. Pagani, G. P. Rossi, and L. Valerio. An opportunistic platform for android-based mobile devices. In *Proceedings of the Second International Workshop on Mobile Opportunistic Networking*, MobiOpp '10, pages 191–193, New York, NY, USA, 2010. ACM.

[54] Muthian Sivathanu and Vijayan Prabhakaran and Florentina I. Popovici and Timothy E. Denehy and Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *Proceedings of the 2nd USENIX Symposium on File and Storage Technologies (FAST '03)*, pages 73–88, 2003.

[55] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, SOSP '97, pages 276–287, New York, NY, USA, 1997. ACM.

[56] RedLicense Labs. RL Benchmark: SQLite. `https://market.android.com/details?id=com.redlicense.benchmark.sqlite`.

[57] Richard Pentin (Summary). Gartner's mobile predictions. `http://ifonlyblog.wordpress.com/2010/01/14/gartners-mobile-predictions/`.

[58] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Trans. Comput. Syst.*, 10(1):26–52, Feb. 1992.

[59] A. Roy, S. M. Rumble, R. Stutsman, P. Levis, D. Mazières, and N. Zeldovich. Energy management in mobile devices with the cinder operating system. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 139–152, New York, NY, USA, 2011. ACM.

[60] Samsung Corp. Samsung ships industrys first multi-chip package with a pram chip for handsets. `http://tinyurl.com/4y9bsds`.

[61] M. Satyanarayanan. Mobile computing: the next decade. In *Proceedings of the 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond*, MCS '10, pages 5:1–5:6, New York, NY, USA, 2010. ACM.

[62] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8:14–23, October 2009.

[63] S. W. Schlosser and G. R. Ganger. MEMS-based storage devices and standard disk interfaces: A square peg in a round hole? pages 87–100.

[64] Ted Tso. Android will be using ext4 starting with Gingerbread. `http://www.linuxfoundation.org/news-media/blogs/browse/2010/12/android-will-be-using-ext4-starting-gingerbread`, Dec. 2010.

[65] N. Tolia, J. Harkes, M. Kozuch, and M. Satyanarayanan. Integrating Portable and Distributed Storage. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, pages 227–238, San Francisco, California, April 2004.

[66] K. Veeraraghavan, J. Flinn, E. B. Nightingale, and B. Noble. qufiles: the right file at the right time. In *Proceedings of the 8th USENIX conference on File and storage technologies*, FAST'10, Berkeley, CA, USA, 2010. USENIX Association.

[67] Z. Wang, F. X. Lin, and L. Zhong. Why are Web Browsers Slow on Smartphones? In *ACM HotMobile '11*, March 2011.

[68] WG802.11 - Wireless LAN Working Group. IEEE STANDARD 802.11n-2009. `http://standards.ieee.org/findstds/standard/802.11n-2009.html`.

Serving Large-scale Batch Computed Data with Project Voldemort

Roshan Sumbaly Jay Kreps Lei Gao Alex Feinberg Chinmay Soman Sam Shah
LinkedIn

Abstract

Current serving systems lack the ability to bulk load massive immutable data sets without affecting serving performance. The performance degradation is largely due to index creation and modification as CPU and memory resources are shared with request serving. We have extended Project Voldemort, a general-purpose distributed storage and serving system inspired by Amazon's Dynamo, to support bulk loading terabytes of read-only data. This extension constructs the index offline, by leveraging the fault tolerance and parallelism of Hadoop. Compared to MySQL, our compact storage format and data deployment pipeline scales to twice the request throughput while maintaining sub 5 ms median latency. At LinkedIn, the largest professional social network, this system has been running in production for more than 2 years and serves many of the data-intensive social features on the site.

1 Introduction

Many social networking and e-commerce web sites contain data-derived features, which usually consist of some data mining application offering insights to the user. Typical features include: "People You May Know," a link prediction system attempting to find other users you might know on the social network (Figure 1a); collaborative filtering, which showcases relationships between pairs of items based on the wisdom of the crowd (Figure 1b); various entity recommendations; and more. LinkedIn, the largest professional social network with, as of writing, more than 135 million members, consists of these and more than 20 other data-derived features.

The feature data cycle in this context consists of a continuous chain of three phases: data collection, processing, and serving. The data collection phase usually involves log consumption, while the processing phase involves running algorithms on the output. Algorithms such as link prediction or nearest-neighbor computation output hundreds of results per user. For example, the "People You May Know" feature on LinkedIn runs on hundreds of terabytes of offline data daily to make these predictions.

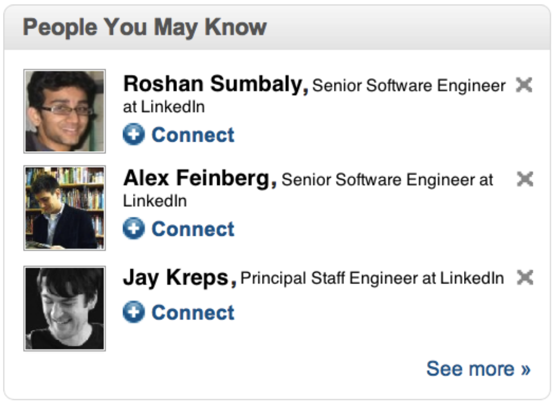
Due to the dynamic nature of the social graph, this derived data changes extremely frequently—requiring an almost complete refresh and bulk load of the data, while continuing to serve existing traffic with minimal additional latency. Naturally, this batch update should complete quickly to engender frequent pushes.

Interestingly, the nature of this complete cycle means that live updates are not necessary and are usually handled by auxiliary data structures. In the collaborative filtering use case, the data is purely static. In the case of "People You May Know", dismissed recommendations (marked by clicking "X") are stored in a separate data store with the difference between the computed recommendations and these dismissals calculated at page load.

This paper presents read-only extensions to Project Voldemort, our key-value solution for the final serving phase of this cycle and discusses how it fits into our feature ecosystem. Voldemort, which was inspired by Amazon's Dynamo [7], was originally designed to support fast online read-writes. Our system leverages a Hadoop elastic batch computing infrastructure to build its index and data files, thereby supporting high throughput for batch refreshes. A custom read-only storage engine plugs into Voldemort's extensible storage layer. The Voldemort infrastructure then provides excellent live serving performance for this batch output—even during data refreshes.

Our system supports quick rollback, where data can be restored to a clean copy, minimizing the time in error if an algorithm should go awry. This helps support fast, iterative development necessary for new feature improvements. The storage data layout also provides the ability to grow horizontally by rebalancing existing data to new nodes without downtime.

Our system supports twice the request throughput versus MySQL while serving read requests with a median latency of less than 5 ms. At LinkedIn, this system has been running for over 2 years, with one of our largest



People You May Know

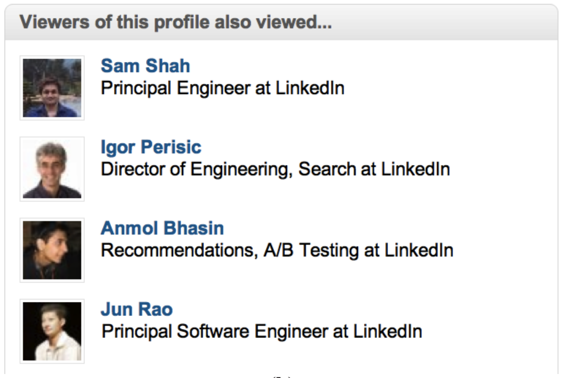
Roshan Sumbaly, Senior Software Engineer at LinkedIn
Connect

Alex Feinberg, Senior Software Engineer at LinkedIn
Connect

Jay Kreps, Principal Staff Engineer at LinkedIn
Connect

See more »

(a)



Viewers of this profile also viewed...

Sam Shah
Principal Engineer at LinkedIn

Igor Perisic
Director of Engineering, Search at LinkedIn

Anmol Bhasin
Recommendations, A/B Testing at LinkedIn

Jun Rao
Principal Software Engineer at LinkedIn

(b)

Figure 1: (a) The "People You May Know" module. (b) An example collaborative filtering module.

clusters loading more than 4 terabytes of new data to the site every day.

The key contributions of this work are:

- A scalable offline index construction, based on MapReduce [6], which produces partitioned data for online consumption
- Complete data cycle to refresh terabytes of data with minimum effect on existing serving latency
- Custom storage format for static data, which leverages the operating system's page cache for cache management

Voldemort and its read-only extensions are open source and are freely available under the Apache 2.0 license.

The rest of the paper is as follows. Section 2 first discusses related work. We then provide an architectural overview of Voldemort in Section 3. We follow with a discussion in Section 4 of existing solutions that we tried, but found insufficient for bulk loading and serving largely static data. Section 5 describes Voldemort's read-only extensions, including our new storage format and how data and indexes are built offline and loaded into the system. Section 6 presents experimental and production results evaluating our solution. We close with a discussion of future directions in Section 7.

2 Related Work

MySQL [16] is a common serving system used in various companies. The two most commonly used MySQL storage engines, MyISAM and InnoDB, provide bulk loading capabilities into a live system with the `LOAD DATA INFILE` statement. MyISAM provides a compact on-disk structure and the ability to delay recreation of the index after the load. However, these benefits come at the expense of requiring considerable memory to maintain a special tree-like cache during bulk loading. Additionally, the MyISAM storage engine locks the complete table for the duration of the load, resulting in queued requests. In comparison, InnoDB supports row-level locking, but its on-disk structure requires considerable disk space and its bulk loading is orders of magnitude slower than MyISAM.

Considerable work has been done to add bulk loading ability to new shared nothing [22] cluster databases similar to Voldemort. Silberstein et al. [19] introduce the problem of bulk insertion into range-partitioned tables in PNUTS [4], which tries to optimize data movement between machines and total transfer time by adding an extra planning phase to gather statistics and prepare the system for the incoming workload. In an extension of that work [20], Hadoop is used to batch insert data into PNUTS in the reduce phase. Both of these approaches optimize the time for data loading into the live system, but incur latency degradation on live serving due to multi-tenant issues with sharing CPU and memory during the full loading process. This is a significant problem with very large data sets, which even after optimizations, might take hours to load.

Our system alleviates this problem by moving the construction of the indexes to an offline system. MapReduce [6] has been used for this offline construction in various search systems [14]. These search layers trigger builds on Hadoop to generate indexes, and on completion, pull the indexes to serve search requests.

This approach has also been extended to various databases. Konstantinou et al. [10] and Barbuzzi et al. [2] suggest building HFiles offline in Hadoop, then shipping them to HBase [9], an open source database modeled after BigTable [3]. These works do not explore the data pipeline, particularly data refreshes and rollback.

The overall architecture of Voldemort was inspired from various DHT storage systems. Unlike the previous DHT systems, such as Chord [21], which provide $O(log\ N)$ lookup, Voldemort's lookups are $O(1)$ because the complete cluster topology is stored on every node. This information allows clients to bootstrap from a random node and direct requests to exact destination nodes. Similar to Dynamo [7], Voldemort also supports per tuple-based replication for availability purposes. Updating replicas is easy in the batch scenario because they are precomputed and loaded into the Voldemort cluster at once. The novelty of Voldemort, compared to Dynamo, is our custom storage engine for bulk-loaded data sets.

3 Project Voldemort

A Voldemort *cluster* can contain multiple *nodes*, each with a unique identifier. A physical host can run multiple nodes, though at LinkedIn we maintain a one-to-one mapping. All nodes in the cluster have the same number of *stores*, which correspond to database tables. General usage patterns have shown that a site-facing feature can map to one or more stores. For example, a feature dealing with group recommendations will map to two stores: one recording a member id to recommended group ids and another recording a group id to its corresponding description. Every store has the following list of configurable parameters, which are identical to Dynamo's parameters:

- *Replication factor* (N): Number of nodes which each key-value tuple is replicated.
- *Required reads* (R): Number of nodes Voldemort reads from, in parallel, during a get before declaring a success.
- *Required writes* (W): Number of node responses Voldemort blocks for, before declaring success during a put.
- *Key/Value serialization and compression*: Voldemort can have different serialization schemas for key and value. For the custom batch data use case, Voldemort uses a custom binary JSON format. Voldemort also supports per tuple-based compression. Serialization and compression is completely handled by a component that resides on the client side with the server only dealing with raw byte arrays.
- *Storage engine type*: Voldemort supports various read-write storage engine formats: Berkeley DB Java Edition [15] and MySQL [16]. Voldemort also supports a custom read-only storage engine for bulk-loaded data.

Every node in the cluster stores the same 2 pieces of metadata: the complete cluster topology and the store definitions.

Voldemort has a pluggable architecture, as shown in Figure 2. Each box represents a module, all of which share the same code interface. Each module has exactly one functionality, making it easy to interchange modules. For example, we can have the routing module on either the client side or the server side. Functional separation at the module level also allows us to easily mock these modules for testing purposes—for example, a mocked up storage engine backed by a hash map for unit tests.

Many of our modules have been inspired by the original Dynamo paper. Starting from the top of the Voldemort stack, our client has a simple get and put API. Every tuple is replicated for availability, with each value having

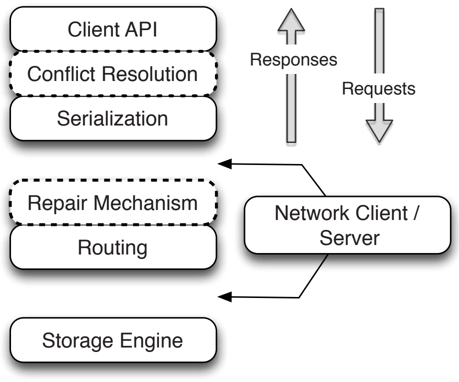
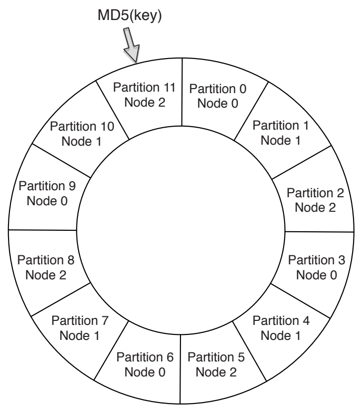


Figure 2: Voldemort architecture containing modules for a single client and server. The dotted modules are not used by the read-only storage engine.



Partition hashed to	Preference list		
	N0	N1	N2
0	***0***	1	
1		***1***	2
2	3		***2***
3	***3***	4	
4		***4***	5
5	6		***5***
6	***6***	7	
7		***7***	8
8	9		***8***
9	***9***	10	
10		***10***	11
11	0		***11***

Figure 3: Simple hash ring cluster topology for 3 nodes and 12 partitions. The preference list generation for a key hashing to partition 11, for a store with N=2, would jump the ring clockwise to place the other $N-1$=1 replica on partition 0. The table shows the preference list generated for every hashed partition. The primary partitions have been highlighted in bold.

vector clock [11] versioning. The "conflict resolution" and "repair mechanism" layer, used only by the read-write storage engines, deal with inconsistent replicas. This does not apply to read-only stores because Voldemort updates all the replicas of a key in a store at once, keeping them in sync.

The "routing" module deals with partitioning as well as replication. Our partitioning scheme is similar to Dynamo's, wherein Voldemort splits the hash ring into equal size partitions, assigns them unique ids, and then maps them to nodes. This ring is then shared with all the stores; that is, changes in the mapping require changes to all the stores. To generate the *preference list* (the list of partition ids where the replicas will be stored), we first hash the key (using MD5) to a range belonging to a partition and then continue jumping the ring clockwise to find $N-1$ partitions belonging to different nodes. For example, for a store with N=2 and partition mapping as shown in Figure 3, the preference list for a key hashing to partition 11 will be (Partition 11, Partition 0).

The last module, the pluggable storage layer, has the same *get* and *put* functions, along with the ability to stream data out. In addition to running the full stack from Figure 2, every node also runs an administrative service that allows the execution of following privileged commands: add or remove a store, stream data out, and trigger read-only store operations.

Voldemort supports two routing modes: server-side and client-side routing. Client-side routing, the more commonly used routing strategy, requires an initial "bootstrap" step, wherein it retrieves the metadata required for routing (cluster topology and store definitions) by load balancing to a random node. Once the metadata has been retrieved by the client, one fewer hop is necessary compared to server-side routing, because the replica locations can be calculated on the fly. However, as we will further explain in Section 5.7, client-side routing makes rebalancing of data complicated, because we now need a mechanism to update the cluster topology metadata on the live clients.

4 Alternative Approaches

Before we started building our own custom storage engine, we decided to evaluate the existing read-write storage engines supported in Voldemort, namely, MySQL and Berkeley DB. Our criteria for success was the ability to bulk load massive data sets with minimal disk space overhead, while still serving live traffic.

4.1 Shortcomings of Alternative Approaches

The first approach we tried was to perform multiple *put* requests. This naïve approach is problematic as every request results in an incremental change to the underlying index structure (in most cases, a B+ tree), which in turn, results in many disk seeks. To solve this problem, MySQL provides a `LOAD DATA` statement that tries to bulk update the underlying index. Unfortunately, using this statement for the MyISAM storage engine locks the entire table. InnoDB instead executes this statement with row-level locking, but experiences substantial disk space overhead for every tuple. However, to achieve MyISAM-like bulk loading performance, InnoDB prefers data ordered by primary key. Achieving fast load times with low space overhead in Berkeley DB requires several manual and non-scalable configuration changes, such as shutting down cleaner and checkpointer threads.

The next solution we explored was to bulk load into a different MySQL table on the same cluster and use views to transparently swap to the new table. We used the MyISAM storage engine, opting to skip InnoDB due to the large space requirements. This approach solves the locking problem, but still hurts serving latency during the load due to pressure on shared CPU and memory resources.

We then tried completely offloading the index construction to another system as building the index on the serving system has isolation problems. We leveraged the fact that MyISAM allows copying of database files from another node into a live database directory, automatically making it available for serving. We bulk load to a separate cluster and then copy the resulting database files over to the live cluster. This two-step approach requires the extra maintenance cost of a separate MySQL cluster with exactly the same number of nodes as the live one. Additionally, the inability to load compressed data in the bulk load phase means data is copied multiple times between nodes: first, as a flat file to the bulk load cluster; then as an internal copy during the `LOAD` statement; and finally, as a raw database file copy to the actual live database. These copies make the load more time-consuming.

The previous solution was not ideal, due to its dependency on redundant MySQL servers and the resulting vulnerability to failure downtime. To address this shortcoming, the next attempted approach used the inherent fault tolerance and parallelism of Hadoop and built individual node/partition-level data stores, which could be transferred to Voldemort for serving. A Hadoop job reads data from a source in HDFS [18], repartitions it on a per-node basis, and finally writes the data to individual storage engines (for example, Berkeley DB) on the local filesystem of the reducer phase Hadoop nodes. The number of reducers equals the number of Voldemort nodes, but could have easily been further split on a per-partition basis. This data is then read from the local filesystem and copied onto HDFS, where it can be fetched by Voldemort. The benefit of this approach is that it leverages Hadoop's parallelism to build the indexes offline; however, it suffers from an extra copy from the local filesystem on the reducer nodes to HDFS, which can become a bottleneck with terabytes of data.

4.2 Requirements

The lack of off-the-shelf solutions, along with the inefficiencies of the previous experiments, motivated the building of a new storage engine and deployment pipeline with the following properties.

- *Minimal performance impact on live requests*: The incoming *get* requests to the live store must not be impacted during the bulk load. There is a trade-off between modifying the current index on the live server and a fast bulk load—quicker bulk loads result in increased I/O, which in turn hurts performance. As a result, we should completely rebuild the index offline and also throttle fetches to Voldemort.
- *Fault tolerance and scalability*: Every step of the data load pipeline should handle failures and also

scale horizontally to support future expansion without downtime.

- *Rollback capability*: The general trend we notice in our business is that incorrect or incomplete data due to algorithm changes or source data problems needs immediate remediation. In such scenarios, running a long batch load job to repopulate correct data is not acceptable. To minimize the time in error, our storage engine must support very fast rollback to a previous good state.
- *Ability to handle large data sets*: The easy access to scalable computing through Hadoop, along with the growing use of complex algorithms has resulted in large data sets being used as part of many core products. Classic examples of this, in the context of social networks, include storing relationships between a pair of users, or between users and an entity. When dealing with millions of users, these pairs can easily reach billions of tuples, motivating our storage engine to support terabytes of data and perform well under a large data to memory ratio.

5 Read-only Extensions

To satisfy the requirements laid out in Section 4.2, we built a new data deployment pipeline as shown in Figure 4. We use the existing Voldemort architecture to plug in a new storage engine with a compact custom format (Section 5.1). For many of LinkedIn's user-facing features, data is generated by algorithms run on Hadoop. For example, the "People You May Know" feature runs a complex series of Hadoop jobs on log data. We thus leverage Hadoop as the computation layer for building the index as its MapReduce component handles failures while HDFS replication provides availability. After the algorithm's computation completes, a driver program coordinates a refresh of the data. As shown in steps 1 and 2 in Figure 4, it triggers a build of the output data in our custom storage format and stores it on HDFS (Section 5.2). This data is kept in versioned format (Section 5.3) after being fetched by Voldemort nodes in parallel (Section 5.4), as demonstrated in steps 3 and 4. Once fetched and swapped in, as displayed in steps 5 and 6, the data on the Voldemort nodes is ready for serving (Section 5.5). This section describes this procedure in detail. We also discuss real world production scenarios such as data schema changes (Section 5.6) and the no-downtime addition of new nodes (Section 5.7).

5.1 Storage Format

Many storage formats try to build data structures that keep the data memory resident in the process's address space, ignoring the effects of the operating system's page cache. The several orders of magnitude latency gap between page cache and disk means that most of the real

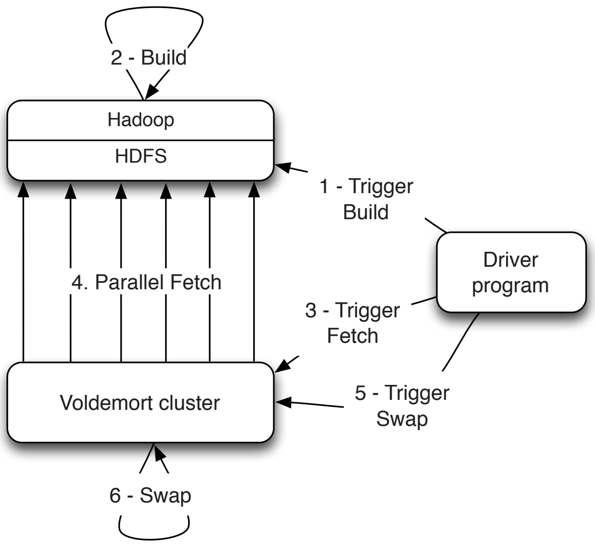


Figure 4: Steps involved in the complete data deployment pipeline. The components involved include Hadoop, HDFS, Voldemort, and a driver program coordinating the full process. The "build" step works on the output of the algorithm's job pipeline.

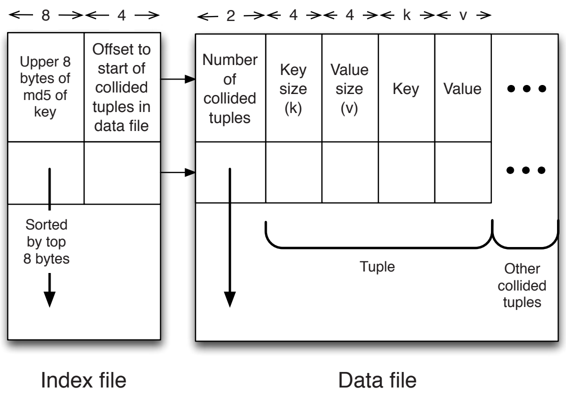


Figure 5: Read-only data is split into multiple chunk buckets, each of which is further split into multiple chunk sets. A chunk set contains an index file and a data file. The diagram shows the data layout in these files. The numbers at the top are sizes in bytes.

performance benefit by maintaining our own structure is for elements already in the page cache. In fact, this custom structure may even start taking memory away from the page cache. This potential interference motivated the need for our storage engine to exploit the page cache instead of maintaining our own complex heap-based data structure. Because our data is immutable, Voldemort memory maps the entire index into the address space. Additionally, because Voldemort is written in Java and runs on the JVM, delegating the memory management to the operating system eases garbage collection tuning.

To take advantage of the parallelism in Hadoop during generation, we split the input data destined for a particular node into multiple *chunk buckets*, which in turn are split into multiple *chunk sets*. Generation of multiple chunk

Node Id	Chunk buckets
0	0_0, 3_0, 6_0, 9_0, 2_1, 5_1, 8_1, 11_1
1	1_0, 4_0, 7_0, 10_0, 0_1, 3_1, 6_1, 9_1
2	2_0, 5_0, 8_0, 11_0, 1_1, 4_1, 7_1, 10_1

Table 1: Every Voldemort node is responsible for chunk buckets based on the primary partition and replica id. This table shows the node id to chunk bucket mapping for the cluster topology defined in Figure 3.

sets can then be done independently and in parallel. A chunk bucket is defined by the primary partition id and replica id, thereby giving it a unique identifier across all nodes. For a store with N=2, the replica id would be either 0 for the primary replica or 1 for the secondary replica. For example, the hashed key in Figure 3 would fall into buckets 11_0 (on node 2) and 11_1 (on node 0). Table 1 summarizes the various chunk buckets for a store with N=2 and cluster topology as shown in Figure 3. Our initial design had started with the simpler design of having one chunk bucket per-node (that is, multiple chunk sets stored on a node with no knowledge of partition/replica), but the current smaller granularity is necessary to aid in rebalancing (Section 5.7).

The number of chunk sets per bucket is decided during generation on the Hadoop side. The default value is one chunk set per bucket, but can be increased by the store owner for more parallelism. The only limitation is that a very large value for this parameter would result in multiple small-sized files—a scenario that HDFS does not handle efficiently. As shown in Figure 5, a chunk set includes a data file and an index file. The standard naming convention for all our chunk sets is *partition id_replica id_chunk set id*.{data, index} where *partition id* is the id of the primary partition, *replica id* is a number between 0 to $N-1$, and *chunk set id* is a number between 0 to the predefined number of sets per bucket-1.

The index file is a compact structure containing the sorted upper 8 bytes of the MD5 of the key followed by the 4 byte offset of the corresponding value in the data file. This simple sorted structure allows us to leverage Hadoop's ability to return sorted data in the reducers. Further, preliminary tests also showed that the index files were generally orders of magnitude smaller than the data files and hence, could fit into the page cache. The use of MD5, instead of any other hash function yielding uniformly distributed values, was an optimization to reuse the calculation from the generation of the preference list.

We had initially started by using the full 16 bytes of the MD5 signature, but saw performance problems as the number of stores grew. In particular, the indexes for all stores were not page cache resident, and thrashing behavior was seen for certain stores due to other high-throughput stores. To alleviate this problem, we needed to cut down on the amount of data being memory mapped, which could be achieved by reducing the available key-space and accepting collisions in the data file.

Our optimization to decrease key-space can be mapped to the classic birthday paradox: if we want to retrieve n random integers from a uniform distribution of range $[1, x]$, the probability that at least 2 numbers are the same is:

$$1 - e^{\frac{-n(n-1)}{2x}} \tag{1}$$

Mapping this to our common scenario of stores keyed by member id, n is our 135 million member user base, while the initial value of x is $2^{128} - 1$ (16 bytes of MD5). The probability of collision in this scenario is close to 0. A key-space of 4 bytes (that is, 32 bits) yields an extremely high collision probability of:

$$1 - e^{\frac{(-135*10^6*(135*10^6-1)}{2*(2^{32}-1)}} \sim 1 \tag{2}$$

Instead, a compromise of 8 bytes (that is, 64 bits) produces:

$$1 - e^{\frac{(-135*10^6*(135*10^6-1)}{2*(2^{64}-1)}} < 0.0004 \tag{3}$$

The probability of more than one collision is even smaller. As a result, by decreasing the number of bytes of the MD5 of the key, we were able to cut down the index size by 40%, allowing more stores to fit into the page cache. The key-space size is an optional parameter the store owner can set depending on the semantics of the data. Unfortunately, this optimization came at the expense of having to save the keys in the data file to use for lookups and handle rare collisions.

The data file is also a very highly-packed structure where we store the number of collided tuples followed by a list of collided tuples *(key size, value size, key, value)*. The order of these multiple lists is the same as the corresponding 8 bytes of MD5 of key in the index file. Here, we need to store the key bytes instead of the MD5 in the tuples to distinguish collided tuples during reads.

5.2 Chunk Set Generation

Construction of the chunk sets for all the Voldemort nodes is a single Hadoop job; the pseudo-code representation is shown in Figure 6. The Hadoop job takes as its input the number of chunk sets per bucket, cluster topology, store definition, and the input data location on HDFS. The job then takes care of replication and partitioning, finally saving the data into separate node-based directories.

At a high level, the mapper phase deals with the partitioning of the data depending on the routing strategy; the partitioner phase redirects the key to the correct reducer and the reducer phase deals with writing the data to a single chunk set. Due to Hadoop's generic `InputFormat` mechanics, any source data can be converted to Voldemort's serialization format. The mapper phase emits the

```
Global Input: Num Chunk Sets: Number of chunk sets per bucket
Global Input: Replication Factor: Tuple replicas for the store
Global Input: Cluster: The cluster topology
Function: TopBytes(x,n): Returns top n bytes of x
Function: MD5(x): Returns MD5 of x
Function: PreferenceList(x): Partition list for key x
Function: Size(x): Return size in bytes
Function: Make*(x): Convert x to Voldemort serialization

Input: K/V: Key/value from HDFS files
Data: K'/V': Transformed key/value into Voldemort serialization
map (K, V)
    K' ← MakeKey(K)
    V' ← MakeValue(V)
    Replica Id ← 0
    MD5K' ← MD5(K')
    KOut ← TopBytes( MD5K', 8 )
    foreach Partition Id ∈ PreferenceList(MD5K') do
        Node Id ← PartitionToNode(Partition Id)
        emit(KOut, [Node Id, Partition Id, Replica Id, K', V'])
        Replica Id ← Replica Id + 1
    end
end
Input: K: Top 8 bytes of MD5 of Voldemort key
Input: V: [Node Id, Partition Id, Replica Id, K', V']
partition (K, V): Integer
    Chunk Set Id ← TopBytes( MD5(V.K'), Size(Integer) )
                            % Num Chunk Sets
    Bucket Id ← V.Partition Id * Replication Factor +
                            V.Replica Id
    return Bucket Id * Num Chunk Sets + Chunk Set Id
end
Input: K/V: Same as partitioner
Data: Position: Continuous offset into data file. Initialized to 0
reduce (K, Iterator<V> Values)
    WriteIndexFile(K)
    WriteIndexFile(Position)
    WriteDataFile(Values.length)
    Position += Size(Short)
    foreach V ∈ Values do
        WriteDataFile( Size(V.K') )
        WriteDataFile( Size(V.V') )
        WriteDataFile(V.K')
        WriteDataFile(V.V')
        Position += Size(V.K') + Size(V.V') + Size(2*Integer)
    end
end
```

Figure 6: MapReduce pseudo-code used for chunk set generation.

upper 8 bytes of the MD5 of the Voldemort key N times as the map phase key with the map phase value equal to a grouped tuple of node id, partition id, replica id, and the Voldemort key and value.

The custom partitioner generates the chunk set id within a chunk bucket from this key. Due to the fair distribution of MD5, we partition the data destined for a bucket into sets with a mod of the 4 bytes of MD5 by the predefined number of chunk sets per bucket. This generated chunk set id, along with the partition id and replication factor of the store, is used to route the data further to the correct reducer.

Finally, every reducer is responsible for a single chunk set, meaning that by having more chunk sets, build phase parallelism can be increased. Hadoop automatically sorts input based on the key in the reduce phase, so data arrives in the order necessary for the index and data files, which can be constructed as simple appends on HDFS with no extra processing required. The data layout on HDFS is a directory for each Voldemort node, with the nomenclature of `node-id`.

5.3 Data Versioning

Before we describe how the generated chunk set files are copied from HDFS, it is essential to understand their storage layout on the Voldemort nodes. This layout is crucial because one of our requirements is the ability to perform instantaneous rollback of data. That is, every time a new copy of the complete data set is created, the system needs to demote the previous copy to an earlier state.

Every store is represented by a directory, which in turn contains directories corresponding to "versions" of the data. A symbolic link per store is used to point to the current serving version directory. Because the data in all version directories except the serving one is inactive, we are not affecting page cache usage and latency. Also, with disks becoming cheaper and providing very fast sequential writes compared to random reads, keeping these previous copies (the number of which is configurable) is beneficial for quick rollback. Every version directory (named `version-no`) has a configurable number associated with it, which should monotonically increase with every new fetch. A commonly used example for the version number is the timestamp of push.

Swapping in a new data version on a single node is done as follows: copy into a new version directory, close the current set of active chunk set files, open the chunk set files from the new version, memory map all the index files, and change the symbolic link to the new version. The entire operation is coordinated using a read-write lock. A rollback follows the same sequence of steps, except that files are opened in an older version directory. Both of these operations are very fast as they are purely metadata operations: no data reads take place.

5.4 Data Load

Figure 4 shows the complete data loading and swapping process for an individual store. Multiple stores can run this entire process concurrently.

The initiator of this complete construction is a standalone driver program that constructs, fetches, and swaps the data. This program starts the process by triggering the Hadoop job described in Section 5.2. The job generates the data on a per-node basis and stores it in HDFS. While streaming the data to HDFS, the Hadoop job also calculates a checksum on a per-node basis by storing a running MD5 on the individual MD5s of all the chunk set files.

Once the Hadoop job is complete, the driver triggers a fetch request on all the Voldemort nodes. This request is received by each node's "administrative service," which then initiates a parallel fetch from HDFS into its respective new version directory. While the data is being streamed from HDFS, the checksum is validated with the checksum from the build step. Voldemort uses a pull model, rather than a push model, as it allows throttling of this fetch in case of latency spikes.

After the data is available on each node in their new version directory, the driver triggers a swap operation (described in Section 5.3) on all nodes. On one of LinkedIn's largest clusters, described in Table 3, this complete operation takes around 0.012 ms on average with the worst swap time of around 0.050 ms. Also, to provide global atomic semantics, the driver ensures that all the nodes have successfully swapped their data, rolling back the successful swaps in case of any other swap failures.

5.5 Retrieval

To find a key, the client generates the preference list and directs the request to the individual nodes. The following is a sketch of the algorithm to find data once it reaches a particular node.

1. Calculate the MD5 of the key.
2. Generate the (a) primary partition id, (b) replica id (the replica being searched when querying this node), and (c) chunk set id (the first 4 bytes of MD5 of the key modulo the number of chunk sets per bucket).
3. Find the corresponding active chunk set files (a data file and an index file) using the 3 variables from the previous step.
4. Perform a search using the top 8 bytes of MD5 of the key as the search key in the sorted index file. Because there are fixed space requirements for every key (12 bytes: 8 bytes for key and 4 bytes for offset), this search does not require internal pointers within the index file. For example, the data location of the i-th element in the sorted index is simply a jump to the offset $12 \cdot i + 8$.
5. If found, read the corresponding data location from the index file and jump to the location in the data file. Iterate through any potential collided tuples, comparing keys, and return the corresponding value on key match.

The most time-consuming step is to search the index file. A binary search in an index of 1 million keys can result in around 20 key comparisons; if the index file is not cached, then 20 disk seeks are required to read one value. As a small optimization, while fetching the files from HDFS, Voldemort fetches the index files after all data files to aid in keeping the index files in the page cache.

Rather than binary search, another retrieval strategy for sorted disk files is interpolation search [17]. This search strategy uses the key distribution to predict the approximate location of the key, rather than halving the search space for every iteration. Interpolation search works well for uniformly distributed keys, dropping the search complexity from $O(log\ N)$ to $O(log\ log\ N)$. This helps in the uncached scenario by reducing the number of disk seeks.

We also evaluated other strategies like Fast [12] and Pegasus [8]. As proved in Manolopoulos and Poulakas [13], most of these are better suited for non-uniform distributions. As MD5 (and its subsets) provides a fairly representative uniform distribution, there will be minimal speedup from these techniques.

5.6 Schema Upgrades

As product features evolve, there are bound to be changes to the underlying data model. For example, an administrator may want to add a new dimension to a store's value or do a complete non-backwards compatible change from storing an array to a map. Because our data is static and the system does a full refresh, Voldemort supports the ability to change the schema of the key and value without downtime. For the client to transparently handle this change, the binary JSON serialization format adds a special version byte during serialization. The mapping of version byte to schema is saved in the store definition metadata. The updated store definition metadata can be propagated to clients by forcing a rebootstrap. Introduction of a new schema after a push is now discovered by the client during deserialization as it can look up the new information after reading the version byte. Similarly, during rollback, the client toggles to an older version of schema and is able to read the data with no downtime.

5.7 Rebalancing

Over time as new stores get added to the cluster, the disk to memory ratio increases beyond initial capacity planning, resulting in increased read latency. Our data being static, the naïve approach of starting a new larger cluster, repushing the data, and switching clients does not scale as it requires massive coordination of multiple clients communicating with many stores.

This necessitates the need to transparently and incrementally add capacity to the cluster independent of data pushes. The rebalancing feature allows us to add new nodes to a live cluster without downtime. This feature was initially written for read-write stores but easily fits into the read-only cycle due to the static nature and fine-grained replication of the data. Our smallest unit of rebalancing is a partition. In other words, the addition of a new node translates to giving the ownership of some partitions to that node. The rebalancing process is run by a tool that coordinates the full process.

The following describes the rebalancing strategy during the addition of a new node. First, the rebalancing tool is provided with the future cluster topology metadata, and with this data, it generates a list of all primary partitions that need to be moved. The tool moves partitions in small batches so as to checkpoint and not refetch too much data in case of failure.

For every small batch of primary partitions, the system generates an intermediate cluster topology metadata, which is the current cluster topology plus changes in ownership of the batch of partitions moved. Voldemort must take care of all secondary replica movements that might be required due to the primary partition movement. A plan is generated that lists the set of donating and stealing node-id pairs along with the chunk buckets being moved. With this plan, the rebalancing tool starts asynchronous processes (through the administrative service) on the stealer nodes to copy all chunk sets corresponding to the moving chunk buckets from their respective donor nodes. Rebalancing works only on the active version of the data, ignoring the previous versions. During this copying, the nodes go into a "rebalancing state" and are not allowed to swap any new data. Here it is important to note that the granularity of the bucket selected makes this process as simple as copying files. If buckets were defined on a per-node basis (that is, have multiple chunk sets on a per-node basis), the system would have had to iterate over all the keys on the node and find the keys belonging to the moving partition, finally running an extra merge step to coalesce with the live index on the stealer node's end.

Once the fetches are complete, the rebalancing tool updates the intermediate cluster topology on all the nodes while also running the swap operation, described in Section 5.3, for all the stores on the stealer and donor nodes. The entire process repeats for every batch of primary partitions.

The intermediate topology change also needs to be propagated to all the clients. Voldemort propagates this information as a lazy process where the clients still use the old metadata. If they contact a node with a request for a key in a partition that the node is no longer responsible for, a special exception is propagated, which results in a rebootstrap step along with a retry of the previous request.

The rebalancing tool has also been designed to handle failure scenarios elegantly. Failure during a fetch is not a problem as no new data has been swapped. However, failure during the topology change and swap phase on some nodes requires (a) changing the cluster topology to the previous good cluster topology on all nodes and (b) rolling back the data on nodes that had successfully swapped.

Table 2a shows the new preference list generation when a new node is introduced to a cluster with the original partition mapping as in Figure 3. For simplicity, we show

Partition hashed to	Preference list		
	N0	N1	N2
0	***0***	1	
1		***1***	2
2	3		***2***
3	***3***	4	
4		***4***	5
5	6		***5***
6	***6***	7	
7		***7***	8
8	9		***8***
9	***9***	10	
10		***10***	11
11	0		***11***

⇨

Partition hashed to	Preference list			
	N0	N1	N2	N3
0	***0***	1		
1		***1***	2	
2	✕		***2***	3
3	✕	4		***3***
4		***4***	5	
5	6		***5***	
6	***6***	7		
7		***7***	8	
8	9		***8***	
9	***9***	10		
10		***10***	11	
11	0		***11***	

(a)

Stealer Node Id	Donor Node Id	Chunk buckets to steal
3	0	3_0, 2_1

(b)

Table 2: (a) Change of preference list generation after addition of 4th node (node id 3) to the cluster defined by ring in Figure 3. The highlighted cells show how moving partition 3 to this new node results in secondary movement of keys hashing to partition 2. (b) Rebalancing plan generated for addition of a new node.

an imbalanced move of only one partition, partition 3, to the new node 3. Table 2b shows the plan that would be generated during rebalancing. The movement of partition 3 results in secondary movement for partition 2 due to node mapping changes in its preference list.

6 Evaluation

Our evaluation answers the following questions:

- Can the system rapidly deploy new data sets?
- What is the read latency, and does it scale with data size and nodes?
- What is the impact on latency during a new data deployment?

We use a simulated data set where the key is a long integer between 0 and a varying number and the value is a fixed size 1024 byte random string. All tests were run on Linux 2.6.18 machines with Dual CPU (each having 64-bit 8 cores running at 2.67 GHz), 24 GB of RAM, 6 disk RAID-10 array and Gigabit Ethernet. We used Community Edition version 5.0.27 and the MyISAM storage engine for all the MySQL tests.

As the read-only storage engine relies on the operating system's page cache, we allocated only 4 GB JVM heap. Similarly, as MyISAM uses a special key cache for index blocks and the page cache for data blocks, we chose the same 4 GB for *key_buffer_size*.

6.1 Build Times

One of the important goals of Voldemort is rapid data deployment, which means the build and push phase must

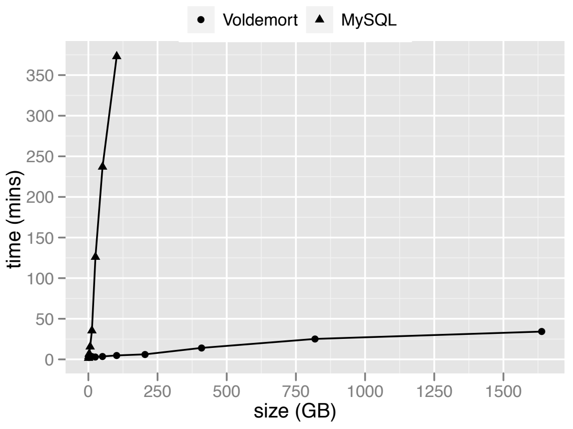


Figure 7: The time to complete the build for the random data set. We vary the input data size by increasing the number of tuples. We terminated the MySQL test early due to prolonged data load time.

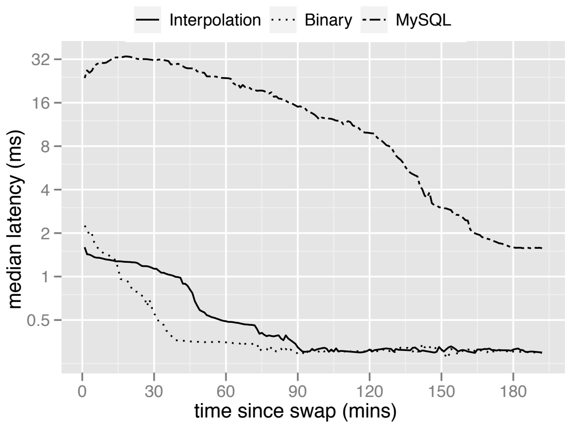


Figure 8: Single node median read latency taken at 1 minute intervals since the swap. The distribution of requests is uniform. The slope of the graph shows the rate of cache warming.

be fast. Push times are entirely dependent on available network bandwidth, so we focus on build times.

We define the build time in the case of Voldemort as the time starting from the first mapper to the end of all reducers. The number of mappers and reducers was fixed across runs to steady the amount of parallelism and generate fixed number of chunk sets per bucket.

In the case of MySQL, the build time is the completion time of the `LOAD DATA INFILE` command on an empty table. This metric ignores the time it took to convert the data to TSV and copy it to the MySQL node. We applied several optimizations to make MySQL faster, including increasing the MySQL bulk insert buffer size and the MyISAM specific sort buffer size to 256 MB each, and also delaying the re-creation of the index to a later time by running the `ALTER TABLE...DISABLE KEYS` statement before the load.

Figure 7 shows the build time as we increased the size of the input data set. As is clearly evident, MySQL exhibits extremely slow build times because it buffers changes to the index before flushing it to the disk. Also, due to the incremental changes required to the index on disk, MySQL does roughly 1.4 times more I/O than our implementation. This factor would increase if we had bulk loaded into a non-empty table.

6.2 Read Latency

Besides rapid data deployments, read latency must be acceptable and the system must scale with the number of nodes. In these experiments, we used 10 million requests with simulated keys following a uniform distribution between 0 to number of tuples in the data set.

We first measure how fast the index loads into the operating system's page cache. We ran tests on a 100 GB data set on a single node and reported the median latency after swap for a continuous stream of uniformly-distributed

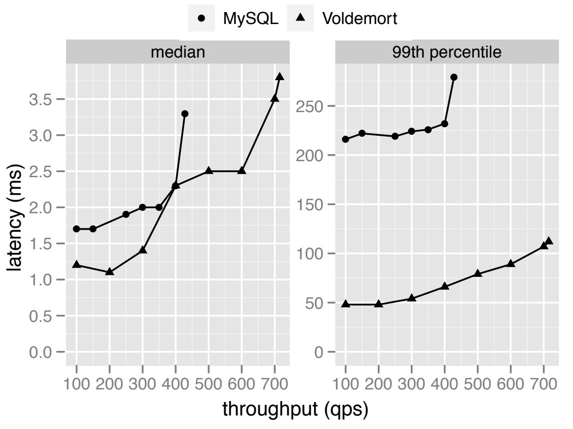


Figure 9: Single node read latency after warming up the cache. This figure shows the change in latency, for uniformly-distributed requests, as we vary the client throughput.

requests. For MySQL, we created a view on an existing table, bulk loaded into a new table, and swapped the view to the new table without stopping the requests. For our read-only storage engine, we used the complete data load process (described in Section 5.4), to swap new data. The single node was configured to have just one partition and one chunk set per bucket. We also compared the binary and interpolation search algorithms for the read-only storage engine.

Figure 8 shows the median latency, at 1 minute intervals, starting from the swap. MySQL starts with a very high median latency due to the uncached index and falls slowly to the stable 1.4 ms mark. Our storage engine starts with low latency because some indexes are already page cache resident, with the fetch phase from HDFS retrieving all index files after the data files. Binary search initially starts with a high median latency compared to interpolation, but the slope of the line is steeper, because

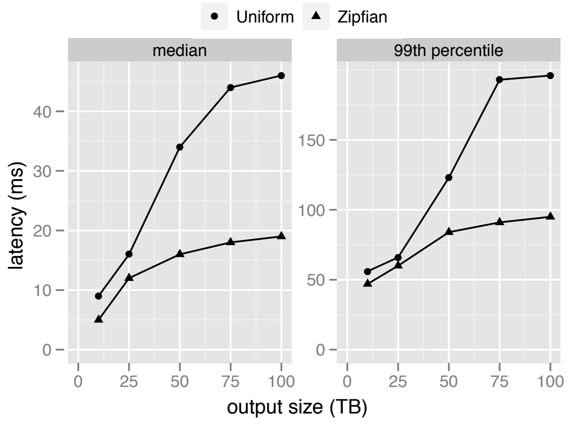


Figure 10: Client-side median latency with varying data size. This test was run on a 32 node cluster with 2 different request distributions.

binary search does an average of 8 lookups, thereby touching more parts of the index; interpolation search performs an average of only 1 lookup. While this results in an initial low read latency, it means that much of the index is uncached in the long run. Our production systems are currently running binary search due to this faster cache warming process. All numbers presented from this point for the read-only storage engine use binary search.

Figure 9 shows a comparison of Voldemort's performance compared to MySQL on the same 100 GB data set for varying throughput. The numbers reported are steady-state latencies; that is, latency reported after the cache is warmed. For comparison, the steady state latency for the read-only storage engine in Figure 9 is around 0.3 ms and is achieved around 90 minutes after the swap. We observed that the time to achieve this steady state, starting from the swap time, is linear in the size of the data set. We increased the client request throughput until the achieved throughput stopped increasing. These results indicate that our implementation scales to roughly twice the number of queries per second while maintaining the same median latency as MySQL.

To test whether our read-only extensions scale with the number of nodes, we evaluated read latency for the same random data set but spread over 32 machines and a store with N=1. The read tests were run for both uniform as well as a Zipfian distribution using YCSB [5], an open source framework for benchmarking cloud data serving systems, with the number of clients fixed at 100 threads. The Zipfian distribution ensures that some keys are more frequently accessed compared to others, simulating the general site visiting patterns of most websites [1]. Figure 10 shows the overall client-side median latency while varying the data set sizes. Querying for frequently accessed keys naturally aids caching certain sections of the indexes, thereby exhibiting an overall lower latency for Zipfian compared to the uniform distribution. We do not report numbers for a store with N>1 because latency is a function of data size and is independent of the replication factor. The results indicate that the system scales with the data set size and the number of nodes. As the data set size increases, we are decreasing the memory to data ratio, affecting read performance. Reducing latency in this case would require adding memory or additional nodes. Users can tune this ratio to achieve the desired latency versus the necessary hardware footprint.

Number of nodes	25
Total (active + backup) data size per node	940 GB
RAM per node	48 GB
Current active data to memory ratio	$\sim$ 10:1
Total number of stores	123
Replication factor for all stores	2
Largest store size (active)	4.15 TB
Smallest store size (active)	700 KB
Max number of store swaps per day	76

Table 3: Statistics for one of LinkedIn's read-only clusters.

6.3 Production Workloads

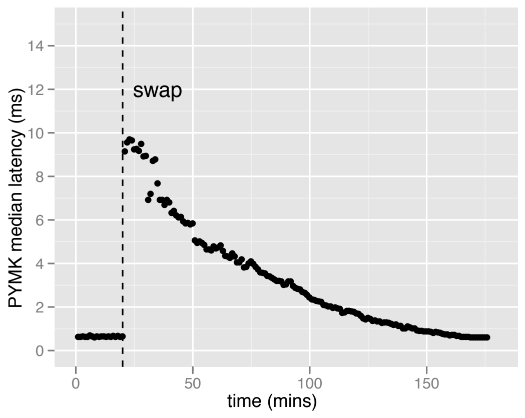
Finally, we show the production performance data for two user-facing features: "People You May Know" (Figure 1a) and collaborative filtering (Figure 1b):

- *People You May Know (PYMK) data set*: Users are presented with a suggested set of other users they might know and would like to connect with. This information is kept as a store where the key is the user's id and the value is a list of integer recommended user ids and a float score.
- *Collaborative filtering (CF) data set*: This feature shows other profiles viewed in the same session as the visited member's profile. The value is a list of two integer ids, a string indicating the entity type, and a float score.

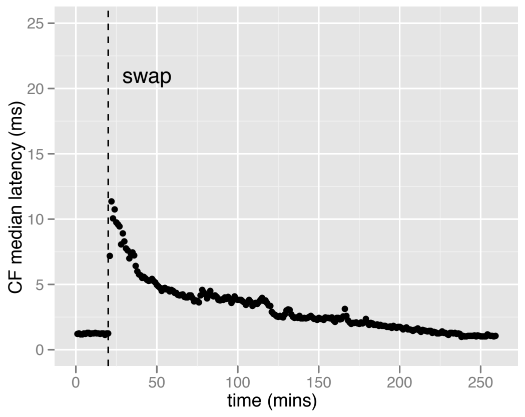
Table 3 shows some statistics for one of LinkedIn's largest clusters. Figure 11 shows the PYMK and CF median client-side read latencies as a function of time since a swap on this cluster (both stores use N=2 and R=1) for one high traffic day. CF has a higher latency than that of PYMK primarily because of the larger value size. We see sub-12 ms latency immediately after a swap with relatively quick stabilization to sub-5 ms latency. This low latency post-swap allows us to push updates to these features multiple times per day.

7 Conclusion and Future Work

In this paper, we present a low-latency bulk loading system capable of serving multiple terabytes of data. By moving the index construction offline to a batch system like Hadoop, our serving layer's performance becomes



(a)



(b)

Figure 11: Median client-side read latency for one of LinkedIn's largest production clusters, as described in Table 3, for the (a) PYMK and (b) CF data sets. The dashed line shows the time when the new data set was swapped in.

more stable and reliable. LinkedIn has been successfully running read-only Voldemort clusters for the past 2 years. It has become an integral part of the product ecosystem with various engineers also using it frequently for quick prototyping of features. The complete system is open-source and freely available.

We plan to add other interesting features to the read-only storage pipeline. Over time we have found that during fetches we exhaust the full bandwidth between data centers running Hadoop (in particular HDFS) and Voldemort. We therefore need improvements to the push process to reduce network usage with minimal impact on build time.

To start with, we are exploring incremental loads. This can be done by generating data file patches on Hadoop by comparing against the previous data snapshot in HDFS and then applying these on the Voldemort side during the fetch phase. We can send the complete index files because (a) they are relatively small files and (b) we can exploit the operating system caching of these files during the fetch phase. This capability has seen few use cases until recently as most of our stores back recommendation features where the delta between data pushes is prohibitively large. Another important feature to save inter-data center bandwidth is the ability to only fetch one replica of the data from HDFS and then propagate it among the Voldemort nodes.

Finally, we are investigating additional index structures that could improve lookup speed and that can easily be built in Hadoop. In particular, cache-oblivious trees, such as van Emde Boas trees [23], require no page size knowledge for optimal cache performance.

Acknowledgements

Bhupesh Bansal and Elias Torres worked on an early version of this project. Thanks to Allen Wittenauer, Anil Alluri, and Bob Liu for their Hadoop support. Several people provided helpful feedback on the paper, including Jun Rao, Bhaskar Ghosh, Igor Perisic, Andrea Dutra, Deepa Elizabeth Jacob, the anonymous FAST reviewers, and our shepherd Kimberly Keeton.

References

[1] Lada Adamic and Bernardo Huberman. Zipf's law and the Internet. *Glottometrics*, 3:143–150, 2002.

[2] Antonio Barbuzzi, Pietro Michiardi, Ernst Biersack, and Gennaro Boggia. Parallel Bulk Insertion for Large-scale Analytics Applications. In *Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware (LADIS '10)*, pages 27–31, New York, NY, USA, 2010.

[3] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson Hsieh, Deborah Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, Berkeley, CA, USA, 2006.

[4] Brian Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proceedings of the VLDB Endowment*, 1:1277–1288, August 2008.

[5] Brian Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*, pages 143–154, New York, NY, USA, 2010.

[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation – Volume 6 (OSDI '04)*, Berkeley, CA, USA, 2004.

[7] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. *SIGOPS Operating Systems Review*, 41:205–220, October 2007.

[8] M. Dowell and P. Jarratt. The Pegasus method for computing the root of an equation. *BIT Numerical Mathematics*, 12:503–508, 1972.

[9] Lars George. *HBase: The Definitive Guide*. O'Reilly Media, 2011.

[10] Ioannis Konstantinou, Evangelos Angelou, Dimitrios Tsoumakos, and Nectarios Koziris. Distributed Indexing of Web Scale Datasets for the Cloud. In *Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud (MDAC '10)*, pages 1:1–1:6, New York, NY, USA, 2010.

[11] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21:558–565, July 1978.

[12] Gilbert N. Lewis, Nancy J. Boynton, and F. Warren Burton. Expected Complexity of Fast Search with Uniformly Distributed Data. *Inf. Process. Lett.*, 13 (1):4–7, 1981.

[13] Yannis Manolopoulos and G Poulakas. An adaptation of a rootfinding method to searching ordered disk files revisited. *BIT Numerical Mathematics*, 29: 364–368, June 1989.

[14] Peter Mika. Distributed Indexing for Semantic Search. In *Proceedings of the 3rd International Semantic Search Workshop (SEMSEARCH '10)*, pages 3:1–3:4, New York, NY, USA, 2010.

[15] Michael Olson, Keith Bostic, and Margo Seltzer. Berkeley DB. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '99)*, Berkeley, CA, USA, 1999.

[16] Sasha Pachev. *Understanding MySQL Internals*. O'Reilly Media, 2007.

[17] Yehoshua Perl, Alon Itai, and Haim Avni. Interpolation search—a log log N search. *Communications of the ACM*, 21:550–553, 1978.

[18] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. *Mass Storage Systems and Technologies, IEEE*, 0:1–10, 2010.

[19] Adam Silberstein, Brian Cooper, Utkarsh Srivastava, Erik Vee, Ramana Yerneni, and Raghu Ramakrishnan. Efficient Bulk Insertion into a Distributed Ordered Table. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*, pages 765–778, New York, NY, USA, 2008.

[20] Adam Silberstein, Russell Sears, Wenchao Zhou, and Brian Cooper. A batch of PNUTS: experiences connecting cloud batch and serving systems. In *Proceedings of the 2011 International Conference on Management of Data (SIGMOD '11)*, pages 1101–1112, New York, NY, USA, 2011.

[21] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. *SIGCOMM Computer Communication Review*, 31:149–160, August 2001.

[22] Michael Stonebraker. The Case for Shared Nothing. *Database Engineering*, 9:4–9, 1986.

[23] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Theory of Computing Systems*, 10:99–127, 1976.

BlueSky: A Cloud-Backed File System for the Enterprise

Michael Vrable[*], Stefan Savage, and Geoffrey M. Voelker

Department of Computer Science and Engineering
University of California, San Diego

Abstract

We present BlueSky, a network file system backed by cloud storage. BlueSky stores data persistently in a cloud storage provider such as Amazon S3 or Windows Azure, allowing users to take advantage of the reliability and large storage capacity of cloud providers and avoid the need for dedicated server hardware. Clients access the storage through a proxy running on-site, which caches data to provide lower-latency responses and additional opportunities for optimization. We describe some of the optimizations which are necessary to achieve good performance and low cost, including a log-structured design and a secure in-cloud log cleaner. BlueSky supports multiple protocols—both NFS and CIFS—and is portable to different providers.

1 Introduction

The promise of third-party "cloud computing" services is a trifecta of reduced cost, dynamic scalability, and high availability. While there remains debate about the precise nature and limit of these properties, it is difficult to deny that cloud services offer real utility—evident in the large numbers of production systems now being cloud-hosted via services such as Amazon's AWS and Microsoft's Azure. However, thus far, services hosted in the cloud have largely fallen into two categories: consumer-facing Web applications (e.g., Netflix customer Web site and streaming control) and large-scale data crunching (e.g., Netflix media encoding pipeline).

Little of this activity, however, has driven widespread outsourcing of enterprise computing and storage applications. The reasons for this are many and varied, but they largely reflect the substantial inertia of existing client-server deployments. Enterprises have large capital and operational investments in client software and depend on the familiar performance, availability and security characteristics of traditional server platforms. In essence, cloud computing is not currently a transparent "drop in" replacement for existing services.

[*]Current affiliation: Google. The work in this paper was performed while a student at UC San Diego.

There are also substantive technical challenges to overcome, as the design points for traditional client-server applications (e.g., file systems, databases, etc.) frequently do not mesh well with the services offered by cloud providers. In particular, many such applications are designed to be bandwidth-hungry and latency-sensitive (a reasonable design in a LAN environment), while the remote nature of cloud service naturally increases latency and the cost of bandwidth. Moreover, while cloud services typically export simple interfaces to abstract resources (e.g., "put file" for Amazon's S3), traditional server protocols can encapsulate significantly more functionality. Thus, until such applications are redesigned, much of the latent potential for outsourcing computing and storage services remains untapped. Indeed, at \$115B/year, small and medium business (SMB) expenditures for servers and storage represent an enormous market should these issues be resolved [9]. Even if the eventual evolution is towards hosting all applications in the cloud, it will be many years before such a migration is complete. In the meantime, organizations will need to support a mix of local applications and use of the cloud.

In this paper, we explore an approach for bridging these domains for one particular application: network file service. In particular, we are concerned with the extent to which traditional network file service can be replaced with commodity cloud services. However, our design is purposely constrained by the tremendous investment (both in capital and training) in established file system client software; we take as a given that end-system software will be unchanged. Consequently, we focus on a proxy-based solution, one in which a dedicated proxy server provides the *illusion* of a single traditional file server in an enterprise setting, translating requests into appropriate cloud storage API calls over the Internet.

We explore this approach through a prototype system, called BlueSky, that supports both NFS and CIFS network file system protocols and includes drivers for both the Amazon EC2/S3 environment and Microsoft's Azure. The engineering of such a system faces a number of design challenges, the most obvious of which revolve around performance (i.e., caching, hiding latency, and maximizing the use of Internet bandwidth), but less intuitively also interact strongly with cost. In particular, the interaction between the storage interfaces and fee schedule provided by current cloud service providers conspire to favor large segment-based layout designs (as well as cloud-based file system cleaners). We demonstrate that ignoring these issues can dramatically inflate costs (as much as 30× in our benchmarks) without significantly improving performance. Finally, across a series of benchmarks we demonstrate that, when using such a design, commodity cloud-based storage services can provide performance competitive with local file servers for the capacity and working sets demanded by enterprise workloads, while still accruing the scalability and cost benefits offered by third-party cloud services.

2 Related Work

Network storage systems have engendered a vast literature, much of it focused on the design and performance of traditional client server systems such as NFS, AFS, CIFS, and WAFL [6, 7, 8, 25]. Recently, a range of efforts has considered other structures, including those based on peer-to-peer storage [16] among distributed sets of untrusted servers [12, 13] which have indirectly informed subsequent cloud-based designs.

Cloud storage is a newer topic, driven by the availability of commodity services from Amazon's S3 and other providers. The elastic nature of cloud storage is reminiscent of the motivation for the Plan 9 write-once file systems [19, 20], although cloud communication overheads and monetary costs argue against a block interface and no storage reclamation. Perhaps the closest academic work to our own is SafeStore [11], which stripes erasure-coded data objects across multiple storage providers, ultimately exploring access via an NFS interface. However, SafeStore is focused clearly on availability, rather than performance or cost, and thus its design decisions are quite different. A similar, albeit more complex system, is DepSky [2], which also focuses strongly on availability, proposing a "cloud of clouds" model to replicate across providers.

At a more abstract level, Chen and Sion create an economic framework for evaluating cloud storage costs and conclude that the computational costs of the cryptographic operations needed to ensure privacy can overwhelm other economic benefits [3]. However, this work predates Intel's AES-NI architecture extension which significantly accelerates data encryption operations.

There have also been a range of non-academic attempts to provide traditional file system interfaces for the key-value storage systems offered by services like Amazon's S3. Most of these install new per-client file system drivers. Exemplars include s3fs [22], which tries to map the file system directly on to S3's storage model (which both changes file system semantics, but also can dramatically increase costs) and ElasticDrive [5], which exports a block-level interface (potentially discarding optimizations that use file-level knowledge such as prefetching).

However, the systems closest to our own are "cloud storage gateways", a new class of storage server that has emerged in the last few years (contemporaneous with our effort). These systems, exemplified by companies such as Nasuni, Cirtas, TwinStrata, StorSimple and Panzura, provide caching network file system proxies (or "gateways") that are, at least on the surface, very similar to our design. Pricing schedules for these systems generally reflect a 2× premium over raw cloud storage costs. While few details of these systems are public, in general they validate the design point we have chosen.

Of commercial cloud storage gateways, Nasuni [17] is perhaps most similar to BlueSky. Nasuni provides a "virtual NAS appliance" (or "filer"), software packaged as a virtual machine which the customer runs on their own hardware—this is very much like the BlueSky proxy software that we build. The Nasuni filer acts as a cache and writes data durably to the cloud. Because Nasuni does not publish implementation details it is not possible to know precisely how similar Nasuni is to BlueSky, though there are some external differences. In terms of cost, Nasuni charges a price based simply on the quantity of disk space consumed (around $0.30/GB/month, depending on the cloud provider)—and not at all a function of data transferred or operations performed. Presumably, Nasuni optimizes their system to reduce the network and per-operation overheads—otherwise those would eat into their profits—but the details of how they do so are unclear, other than by employing caching.

Cirtas [4] builds a cloud gateway as well but sells it in appliance form: Cirtas's Bluejet is a rack-mounted computer which integrates software to cache file system data with storage hardware in a single package. Cirtas thus has a higher up-front cost than Nasuni's product, but is easier to deploy. Panzura [18] provides yet another CIFS/NFS gateway to cloud storage. Unlike BlueSky and the others, Panzura allows multiple customer sites to each run a cloud gateway. Each of these gateways accesses the same underlying file system, so Panzura is particularly appropriate for teams sharing data over a wide area. But again, implementation details are not provided.

TwinStrata [29] and StorSimple [28] implement gateways that present a block-level storage interface, like ElasticDrive, and thus lose many potential file system-level optimizations as well.

In some respects BlueSky acts like a local storage server that backs up data to the cloud—a local NFS server combined with Mozy [15], Cumulus [30], or similar software could provide similar functionality. However, such backup tools may not support a high backup frequency (ensuring data reaches the cloud quickly) and efficient random access to files in the cloud. Further, they treat the local data (rather than the cloud copy) as authoritative, preventing the local server from caching just a subset of the files.

3 Architecture

BlueSky provides service to clients in an enterprise using a transparent proxy-based architecture that stores data persistently on cloud storage providers (Figure 1). The enterprise setting we specifically consider consists of a single proxy cache colocated with enterprise clients, with a relatively high-latency yet high-bandwidth link to cloud storage, with typical office and engineering request workloads to files totaling tens of terabytes. This section discusses the role of the proxy and cloud provider components, as well as the security model supported by BlueSky. Sections 4 and 5 then describe the layout and operation of the BlueSky file system and the BlueSky proxy, respectively.

Cloud storage acts much like another layer in the storage hierarchy. However, it presents new design considerations that, combined, make it distinct from other layers and strongly influence its use as a file service. The high latency to the cloud necessitates aggressive caching close to the enterprise. On the other hand, cloud storage has elastic capacity and provides operation service times independent of spatial locality, thus greatly easing free space management and data layout. Cloud storage interfaces often only support writing complete objects in an operation, preventing the efficient update of just a portion of a stored object. This constraint motivates an append rather than an overwrite model for storing data.

Monetary cost also becomes an explicit metric of optimization: cloud storage capacity might be elastic, but still needs to be parsimoniously managed to minimize storage costs over time [30]. With an append model of storage, garbage collection becomes a necessity. Providers also charge a small cost for each operation. Although slight, costs are sufficiently high to motivate aggregating small objects (metadata and small files) into larger units when writing data. Finally, outsourcing data storage makes security a primary consideration.

3.1 Local Proxy

The central component of BlueSky is a proxy situated between clients and cloud providers. The proxy communicates with clients in an enterprise using a standard network file system protocol, and communicates with cloud providers using a cloud storage protocol. Our prototype supports both the NFS (version 3) and CIFS protocols for clients, and the RESTful protocols for the Amazon S3 and Windows Azure cloud services. Ideally, the proxy

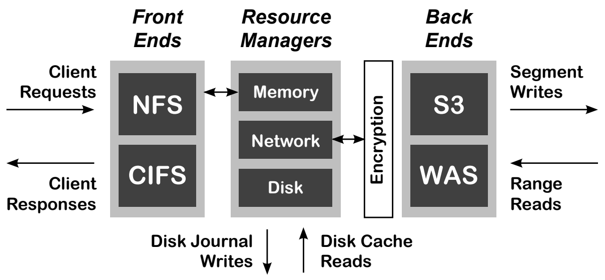


Figure 1: BlueSky architecture.

runs in the same enterprise network as the clients to minimize latency to them. The proxy caches data locally and manages sharing of data among clients without requiring an expensive round-trip to the cloud.

Clients do not require modification since they continue to use standard file-sharing protocols. They mount BlueSky file systems exported by the proxy just as if they were exported from an NFS or CIFS server. Further, the same BlueSky file system can be mounted by any type of client with shared semantics equivalent to Samba.

As described in more detail later, BlueSky lowers cost and improves performance by adopting a log-structured data layout for the file system stored on the cloud provider. A cleaner reclaims storage space by garbage-collecting old log segments which do not contain any live objects, and processing almost-empty segments by copying live data out of old segments into new segments.

As a write-back cache, the BlueSky proxy can fully satisfy client write requests with local network file system performance by writing to its local disk—as long as its cache capacity can absorb periods of write bursts as constrained by the bandwidth the proxy has to the cloud provider (Section 6.5). For read requests, the proxy can provide local performance to the extent that the proxy can cache the working set of the client read workload (Section 6.4).

3.2 Cloud Provider

So that BlueSky can potentially use any cloud provider for persistent storage service, it makes minimal assumptions of the provider; in our experiments, we use both Amazon S3 and the Windows Azure blob service. BlueSky requires only a basic interface supporting `get`, `put`, `list`, and `delete` operations. If the provider also supports a hosting service, BlueSky can co-locate the file system cleaner at the provider to reduce cost and improve cleaning performance.

3.3 Security

Security becomes a key concern with outsourcing critical functionality such as data storage. In designing BlueSky, our goal is to provide high assurances of data confidentiality and integrity. The proxy encrypts all client data before sending it over the network, so the provider cannot read private data. Encryption is at the level of objects (inodes, file blocks, etc.) and not entire log segments. Data stored at the provider also includes integrity checks to detect any tampering by the storage provider.

However, some trust in the cloud provider is unavoidable, particularly for data availability. The provider can always delete or corrupt stored data, rendering it unavailable. These actions could be intentional—e.g., if the provider is malicious—or accidental, for instance due to insufficient redundancy in the face of correlated hardware failures from disasters. Ultimately, the best guard against such problems is through auditing and the use of multiple independent providers [2, 11]. BlueSky could readily incorporate such functionality, but doing so remains outside the scope of our current work.

A buggy or malicious storage provider could also serve stale data. Instead of returning the most recent data, it could return an old copy of a data object that nonetheless has a valid signature (because it was written by the client at an earlier time). By authenticating pointers between objects starting at the root, however, BlueSky prevents a provider from selectively rolling back file data. A provider can only roll back the entire file system to an earlier state, which customers will likely detect.

BlueSky can also take advantage of computation in the cloud for running the file system cleaner. As with storage, we do not want to completely trust the computational service, yet doing so provides a tension in the design. To maintain confidentiality, data encryption keys should not be available on cloud compute nodes. Yet, if cloud compute nodes are used for file system maintenance tasks, the compute nodes must be able to read and manipulate file system data structures. For BlueSky, we make the tradeoff of encrypting file data while leaving the metadata necessary for cleaning the file system unencrypted. As a result, storage providers can understand the layout of the file system, but the data remains confidential and the proxy can still validate its integrity.

In summary, BlueSky provides strong confidentiality and slightly weaker integrity guarantees (some data rollback attacks might be possible but are largely prevented), but must rely on the provider for availability.

4 BlueSky File System

This section describes the BlueSky file system layout. We present the object data structures maintained in the file system and their organization in a log-structured format. We also describe how BlueSky cleans the logs comprising the file system, and how the design conveniently lends itself to providing versioned backups of the data stored in the file system.

4.1 Object Types

BlueSky uses four types of objects for representing data and metadata in its log-structured file system [23] format: data blocks, inodes, inode maps, and checkpoints. These objects are aggregated into log segments for storage. Figure 2 illustrates their relationship in the layout of the file system. On top of this physical layout BlueSky provides standard POSIX file system semantics, including atomic renames and hard links.

Data blocks store file data. Files are broken apart into fixed-size blocks (except the last block may be short). BlueSky uses 32 KB blocks instead of typical disk file system sizes like 4 KB to reduce overhead: block pointers as well as extra header information impose a higher per-block overhead in BlueSky than in an on-disk file system. In the evaluations in Section 6, we show the cost and performance tradeoffs of this decision. Nothing fundamental, however, prevents BlueSky from using variable-size blocks optimized for the access patterns of each file, but we have not implemented this approach.

Inodes for all file types include basic metadata: ownership and access control, timestamps, etc. For regular files, inodes include a list of pointers to data blocks with the file contents. Directory entries are stored inline within the directory inode to reduce the overhead of path traversals. BlueSky does not use indirect blocks for locating file data—inodes directly contain pointers to all data blocks (easy to do since inodes are not fixed-size).

Inode maps list the locations in the log of the most recent version of each inode. Since inodes are not stored at fixed locations, inode maps provide the necessary level of indirection for locating inodes.

A checkpoint object determines the root of a file system snapshot. A checkpoint contains pointers to the locations of the current inode map objects. On initialization the proxy locates the most recent checkpoint by scanning backwards in the log, since the checkpoint is always one of the last objects written. Checkpoints are useful for maintaining file system integrity in the face of proxy failures, for decoupling cleaning and file service, and for providing versioned backup.

4.2 Cloud Log

For each file system, BlueSky maintains a separate log for each writer to the file system. Typically there are two: the proxy managing the file system on behalf of clients and a cleaner that garbage collects overwritten data. Each writer stores its log segments to a separate directory (different key prefix), so writers can make updates to the file system independently.

Each log consists of a number of log segments, and each log segment aggregates multiple objects together into an approximately fixed-size container for storage and transfer. In the current implementation segments are

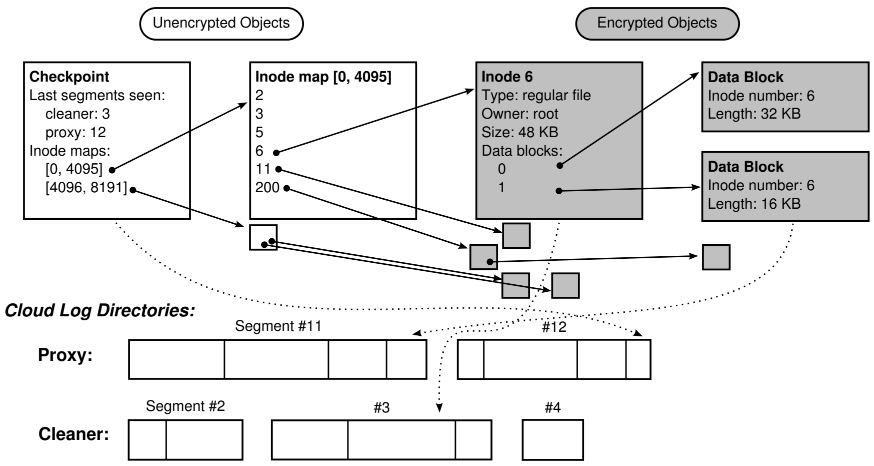


Figure 2: BlueSky filesystem layout. The top portion shows the logical organization. Object pointers are shown with solid arrows. Shaded objects are encrypted (but pointers are always unencrypted). The bottom of the figure illustrates how these log items are packed into segments stored in the cloud.

up to about 4 MB, large enough to avoid the overhead of dealing with many small objects. Though the storage interface requires that each log segment be written in a single operation, typically cloud providers allow partial reads of objects. As a result, BlueSky can read individual objects regardless of segment size. Section 6.6 quantifies the performance benefits of grouping data into segments and of selective reads, and Section 6.7 quantifies their cost benefits.

A monotonically-increasing *sequence number* identifies each log segment within a directory, and a byte *offset* identifies a specific object in the segment. Together, the triple $\langle directory, sequence\ number, offset \rangle$ describes the physical location of each object. Object pointers also include the size of the object; while not required this hint allows BlueSky to quickly issue a read request for the exact bytes needed to fetch the object.

In support of BlueSky's security goals (Section 3.3), file system objects are individually encrypted (with AES) and protected with a keyed message authentication code (HMAC-SHA-256) by the proxy before uploading to the cloud service. Each object contains data with a mix of protections: some data is encrypted and authenticated, some data is authenticated plain-text, and some data is unauthenticated. The keys for encryption and authentication are not shared with the cloud, though we assume that customers keep a safe backup of these keys for disaster recovery. Figure 3 summarizes the fields included in objects.

BlueSky generates a unique identifier (UID) for each object when the object is written into the log. The UID remains constant if an item is simply relocated to a new log position. An object can contain pointers to other objects—for example, an inode pointing to data blocks—and the pointer lists both the UID and the physical location. A cleaner in the cloud can relocate objects and update pointers with the new locations; as long as the UID in the pointer and the object match, the proxy can validate that the data has not been tampered with.

Authenticated: {
Object type
Unique identifier (UID)
Inode number
Encrypted: {Object payload
Object pointers: UIDs

Unauthenticated: Object pointers: Physical locations

Figure 3: Data fields included in most objects.

4.3 Cleaner

As with any log-structured file system, BlueSky requires a file system cleaner to garbage collect data that has been overwritten. Unlike traditional disk-based systems, the elastic nature of cloud storage means that the file system can grow effectively unbounded. Thus, the cleaner is not necessary to make progress when writing out new data, only to reduce storage costs and defragment data for more efficient access.

We designed the BlueSky cleaner so that it can run either at the proxy or on a compute instance within the cloud provider where it has faster, cheaper access to the storage. For example, when running the cleaner in Amazon EC2 and accessing storage in S3, Amazon does not charge for data transfers (though it still charges for operations). A cleaner running in the cloud does not need to be fully trusted—it will need permission to read and write cloud storage, but does not require the file system encryption and authentication keys.

The cleaner runs online with no synchronous interactions with the proxy: clients can continue to access and modify the file system even while the cleaner is running. Conflicting updates to the same objects are later merged by the proxy, as described in Section 5.3.

4.4 Backups

The log-structured design allows BlueSky to integrate file system snapshots for backup purposes easily. In fact, so long as a cleaner is never run, any checkpoint record ever written to the cloud can be used to reconstruct the state of the file system at that point in time. Though not implemented in our prototype, the cleaner or a snapshot tool could record a list of checkpoints to retain and protect all required log segments from deletion. Those segments could also be archived elsewhere for safekeeping.

4.5 Multi-Proxy Access

In the current BlueSky implementation only a single proxy can write to the file system, along with the cleaner which can run in parallel. It would be desirable to have multiple proxies reading from and writing to the same BlueSky file system at the same time—either from a single site, to increase capacity and throughput, or from multiple sites, to optimize latency for geographically-distributed clients.

The support for multiple file system logs in BlueSky should make it easier to add support for multiple concurrent proxies. Two approaches are possible. Similar to Ivy [16], the proxies could be unsynchronized, offering loose consistency guarantees and assuming only a single site updates a file most of the time. When conflicting updates occur in the uncommon case, the system would present the user with multiple file versions to reconcile.

A second approach is to provide stronger consistency by serializing concurrent access to files from multiple proxies. This approach adds the complexity of some type of distributed lock manager to the system. Since cloud storage itself does not provide the necessary locking semantics, a lock manager would either need to run on a cloud compute node or on the proxies (ideally, distributed across the proxies for fault tolerance).

Exploring either option remains future work.

5 BlueSky Proxy

This section describes the design and implementation of the BlueSky proxy, including how it caches data in memory and on disk, manages its network connections to the cloud, and indirectly cooperates with the cleaner.

5.1 Cache Management

The proxy uses its local disk storage to implement a write-back cache. The proxy logs file system write requests from clients (both data and metadata) to a journal on local disk, and ensures that data is safely on disk before telling clients that data is committed. Writes are sent to the cloud asynchronously. Physically, the journal is broken apart into sequentially-numbered files on disk (journal segments) of a few megabytes each.

This write-back caching does mean that in the event of a catastrophic failure of the proxy—if the proxy's storage is lost—that some data may not have been written to the cloud and will be lost. If the local storage is intact no data will be lost; the proxy will replay the changes recorded in the journal. Periodically, the proxy snapshots the file system state, collects new file system objects and any inode map updates into one or more log segments, and uploads those log segments to cloud storage. Our prototype proxy implementation does not currently perform deduplication, and we leave exploring the tradeoffs of such an optimization for future work.

There are tradeoffs in choosing how quickly to flush data to the cloud. Writing data to the cloud quickly minimizes the window for data loss. However, a longer timeout has advantages as well: it enables larger log segment sizes, and it allows overlapping writes to be combined. In the extreme case of short-lived temporary files, no data need be uploaded to the cloud. Currently the BlueSky proxy commits data as frequently as once every five seconds. BlueSky does not start writing a new checkpoint until the previous one completes, so under a heavy write load checkpoints may commit less frequently.

The proxy keeps a cache on disk to satisfy many read requests without going to the cloud; this cache consists of old journal segments and log segments downloaded from cloud storage. Journal and log segments are discarded from the cache using an LRU policy, except that journal segments not yet committed to the cloud are kept pinned in the cache. At most half of the disk cache can be pinned in this way. The proxy sends HTTP byte-range requests to decrease latency and cost when only part of a log segment is needed. It stores partially-downloaded segments as sparse files in the cache.

5.2 Connection Management

The BlueSky storage backends reuse HTTP connections when sending and receiving data from the cloud; the CURL library handles the details of this connection pooling. Separate threads perform each upload or download. BlueSky limits uploads to no more than 32 segments concurrently, to limit contention among TCP sessions and to limit memory usage in the proxy (it buffers each segment entirely in memory before sending).

5.3 Merging System State

As discussed in Section 4.3, the proxy and the cleaner operate independently of each other. When the cleaner runs, it starts from the most recent checkpoint written by

```
**merge_inode**($ino_p$, $ino_c$):
    if $ino_p.id = ino_c.id$:
        return $ino_c$    // No conflicting changes
    // Start with proxy version and merge cleaner changes
    $ino_m \leftarrow ino_p$; $ino_m.id \leftarrow fresh\_uuid()$; $updated \leftarrow false$
    for $i$ in $[0 \ldots num\_blocks(ino_p) - 1]$:
        $b_p \leftarrow ino_p.blocks[i]$; $b_c \leftarrow ino_c.blocks[i]$
        if $b_c.id = b_p.id$ **and** $b_c.loc \neq b_p.loc$:
            // Relocated data by cleaner is current
            $ino_m.blocks.append(b_c)$; $updated \leftarrow true$
        else:    // Take proxy's version of data block
            $ino_m.blocks.append(b_p)$
    return ($ino_m$ if $updated$ else $ino_p$)
```

Figure 4: Pseudocode for the proxy algorithm that merges state for possibly divergent inodes. Subscripts $_p$ and $_c$ indicate state written by the proxy and cleaner, respectively; $_m$ is used for a candidate merged version.

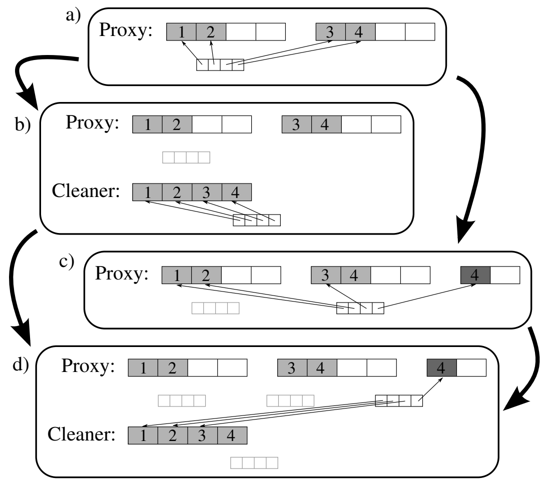


Figure 5: Example of concurrent updates by cleaner and proxy, and the resulting merged state.

the proxy. The cleaner only ever accesses data relative to this file system snapshot, even if the proxy writes additional updates to the cloud. As a result, the proxy and cleaner each may make updates to the same objects (e.g., inodes) in the file system. Since reconciling the updates requires unencrypted access to the objects, the proxy assumes responsibility for merging file system state.

When the cleaner finishes execution, it writes an updated checkpoint record to its log; this checkpoint record identifies the snapshot on which the cleaning was based. When the proxy sees a new checkpoint record from the cleaner, it begins merging updates made by the cleaner with its own updates.

BlueSky does not currently support the general case of merging file system state from many writers, and only supports the special case of merging updates from a single proxy and cleaner. This case is straightforward since only the proxy makes logical changes to the file system and the cleaner merely relocates data. In the worst case, if the proxy has difficulty merging changes by the cleaner it can simply discard the cleaner's changes.

The persistent UIDs for objects can optimize the check for whether merging is needed. If both the proxy and cleaner logs use the same UID for an object, the cleaner's version may be used. The UIDs will differ if the proxy has made any changes to the object, in which case the objects must be merged or the proxy's version used. For data blocks, the proxy's version is always used. For inodes, the proxy merges file data block-by-block according to the algorithm shown in Figure 4. The proxy can similarly use inode map objects directly if possible, or write merged maps if needed.

Figure 5 shows an example of concurrent updates by the cleaner and proxy. State (a) includes a file with four blocks, stored in two segments written by the proxy. At (b) the cleaner runs and relocates the data blocks. Concurrently, in (c) the proxy writes an update to the file, changing the contents of block 4. When the proxy merges state in (d), it accepts the relocated blocks 1–3 written by the cleaner but keeps the updated block 4. At this point, when the cleaner runs again it can garbage collect the two unused proxy segments.

5.4 Implementation

Our BlueSky prototype is implemented primarily in C, with small amounts of C++ and Python. The core BlueSky library, which implements the file system but not any of the front-ends, consists of 8500 lines of code (including comments and whitespace). BlueSky uses GLib for data structures and utility functions, libgcrypt for cryptographic primitives, and libs3 and libcurl for interaction with Amazon S3 and Windows Azure.

Our NFS server consists of another 3000 lines of code, not counting code entirely generated by the rpcgen RPC protocol compiler. The CIFS server builds on top of Samba 4, adding approximately 1800 lines of code in a new backend. These interfaces do not fully implement all file system features such as security and permissions handling, but are sufficient to evaluate the performance of the system. The prototype in-cloud file system cleaner is implemented in just 650 lines of portable Python code and does not depend on the BlueSky core library.

6 Evaluation

In this section we evaluate the BlueSky proxy prototype implementation. We explore performance from the proxy to the cloud, the effect of various design choices on both performance and cost, and how BlueSky performance varies as a function of its ability to cache client working sets for reads and absorb bursts of client writes.

6.1 Experimental Setup

We ran experiments on Dell PowerEdge R200 servers with 2.13 GHz Intel Xeon X3210 (quad-core) processors, a 7200 RPM 80 GB SATA hard drive, and gigabit network connectivity (internal and to the Internet). One machine, with 4 GB of RAM, is used as a load generator. The second machine, with 8 GB of RAM and an additional 1.5 TB 7200 RPM disk drive, acts as a standard file server or a BlueSky proxy. Both servers run Debian testing; the load generator machine is a 32-bit install (required for SPECsfs) while the proxy machine uses a 64-bit operating system. For comparison purposes we also ran a few tests against a commercial NAS filer in production use by our group. We focused our efforts on two providers: Amazon's Simple Storage Service (S3) [1] and Windows Azure storage [14]. For Amazon S3, we looked at both the standard US region (East Coast) as well as S3's West Coast (Northern California) region.

We use the SPECsfs2008 [27] benchmark in many of our performance evaluations. SPECsfs can generate both NFSv3 and CIFS workloads patterned after real-world traces. In these experiments, SPECsfs subjects the server to increasing loads (measured in operations per second) while simultaneously increasing the size of the working set of files accessed. Our use of SPECsfs for research purposes does not follow all rules for fully-compliant benchmark results, but should allow for relative comparisons. System load on the load generator machine remains low, and the load generator is not the bottleneck.

In several of the benchmarks, the load generator machine mounts the BlueSky file system with the standard Linux NFS client. In Section 6.4, we use a synthetic load generator which directly generates NFS read requests (bypassing the kernel NFS client) for better control.

6.2 Cloud Provider Bandwidth

To understand the performance bounds on any implementation and to guide our specific design, we measured the performance our proxy is able to achieve writing data to Amazon S3. Figure 6 shows that the BlueSky proxy has the potential to fully utilize its gigabit link to S3 if it uses large request sizes and parallel TCP connections. The graph shows the total rate at which the proxy could upload data to S3 for a variety of request sizes and number of parallel connections. Network round-trip time from the proxy to the standard S3 region, shown in the graph, is around 30 ms. We do not pipeline requests—we wait for confirmation for each object on a connection before sending another one—so each connection is mostly idle when uploading small objects. Larger objects better utilize the network, but objects of one to a few megabytes are sufficient to capture most gains. A single connection utilizes only a fraction of the total bandwidth, so to fully make use of the network we need multiple parallel

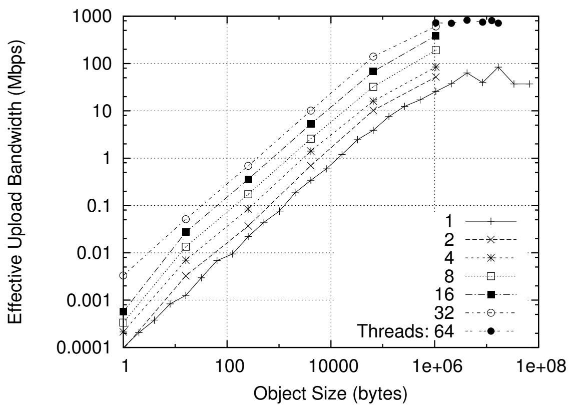


Figure 6: Measured aggregate upload performance to Amazon S3, as a function of the size of the objects uploaded (x-axis) and number of parallel connections made (various curves). A gigabit network link is available. Full use of the link requires parallel uploads of large objects.

TCP connections. These measurements helped inform the choice of 4 MB log segments (Section 4.1) and a pool size of 32 connections (Section 5.2).

The S3 US-West data center is closer to our proxy location and has a correspondingly lower measured round-trip time of 12 ms. The round-trip time to Azure from our location was substantially higher, around 85 ms. Yet network bandwidth was not a bottleneck in either case, with the achievable bandwidth again approaching 1 Gbps. In most benchmarks, we use the Amazon US-West region as the default cloud storage service.

6.3 Impact of Cloud Latency

To underscore the impact latency can have on file system performance, we first run a simple, time-honored benchmark of unpacking and compiling a kernel source tree. We measure the time for three steps: (1) extract the sources for Linux 2.6.37, which consist of roughly 400 MB in 35,000 files (a write-only workload); (2) checksum the contents of all files in the extracted sources (a read-only workload); (3) build an i386 kernel using the default configuration and the `-j4` flag for up to four parallel compiles (a mixed read/write workload). For a range of comparisons, we repeat this experiment on a number of system configurations. In all cases with a remote file server, we flushed the client's cache by unmounting the file system in between steps.

Table 1 shows the timing results of the benchmark steps for the various system configurations. Recall that the network links client↔proxy and proxy↔S3 are both 1 Gbps—the only difference is latency (12 ms from the proxy to BlueSky/S3-West and 30 ms to BlueSky/S3-East). Using a network file system, even locally, adds considerably to the execution time of the benchmark

	Unpack	*Check*	*Compile*
Local file system			
warm client cache	0:30	0:02	3:05
cold client cache		0:27	
Local NFS server			
warm server cache	10:50	0:26	4:23
cold server cache		0:49	
Commercial NAS filer			
warm cache	2:18	3:16	4:32
NFS server in EC2			
warm server cache	65:39	26:26	74:11
BlueSky/S3-West			
warm proxy cache	5:10	0:33	5:50
cold proxy cache		26:12	7:10
full segment		1:49	6:45
BlueSky/S3-East			
warm proxy	5:08	0:35	5:53
cold proxy cache		57:26	8:35
full segment		3:50	8:07

Table 1: Kernel compilation benchmark times for various file server configurations. Steps are (1) unpack sources, (2) checksum sources, (3) build kernel. Times are given in minutes:seconds. Cache flushing and prefetching are only relevant in steps (2) and (3).

compared to a local disk. However, running an NFS server in EC2 compared to running it locally increases execution times by a factor of 6–30× due to the high latency between the client and server and a workload with operations on many small files. In our experiments we use a local Linux NFS server as a baseline. Our commercial NAS filer does give better write performance than a Linux NFS server, likely due in part to better hardware and an NVRAM write cache. Enterprises replacing such filers with BlueSky on generic rack servers would therefore experience a drop in write performance.

The substantial impact latency can have on workload performance motivates the need for a proxy architecture. Since clients interact with the BlueSky proxy with low latency, BlueSky with a warm disk cache is able to achieve performance similar to a local NFS server. (In this case, BlueSky performs slightly better than NFS because its log-structured design is better-optimized for some write-heavy workloads; however, we consider this difference incidental.) With a cold cache, it has to read small files from S3, incurring the latency penalty of reading from the cloud. Ancillary prefetching from fetching full 4 MB log segments when a client requests data in any part of the segment greatly improves performance, in part because this particular benchmark has substantial locality; later on we will see that, in workloads with little locality, full segment fetches hurt performance. However, execution times are still multiples of BlueSky with a warm cache. The differences in latencies between S3-West and S3-East for the cold cache and full segment cases again underscore the sensitivity to cloud latency.

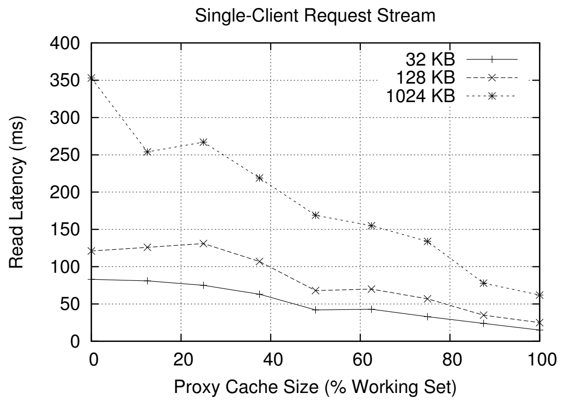


Figure 7: Read latency as a function of working set captured by the proxy. Results are from a single run.

In summary, greatly masking the high latency to cloud storage—even with high-bandwidth connectivity to the storage service—requires a local proxy to minimize latency to clients, while fully masking high cloud latency further requires an effective proxy cache.

6.4 Caching the Working Set

The BlueSky proxy can mask the high latency overhead of accessing data on a cloud service by caching data close to clients. For what kinds of file systems can such a proxy be an effective cache? Ideally, the proxy needs to cache the working set across all clients using the file system to maximize the number of requests that the proxy can satisfy locally. Although a number of factors can make generalizing difficult, previous studies have estimated that clients of a shared network file system typically have a combined working set that is roughly 10% of the entire file system in a day, and less at smaller time scales [24, 31]. For BlueSky to provide acceptable performance, it must have the capacity to hold this working set. As a rough back-of-the-envelope using this conservative daily estimate, a proxy with one commodity 3 TB disk of local storage could capture the daily working set for a 30 TB file system, and five such disks raises the file system size to 150 TB. Many enterprise storage needs fall well within this envelope, so a BlueSky proxy can comfortably capture working sets for such scenarios.

In practice, of course, workloads are dynamic. Even if proxy cache capacity is not an issue, clients shift their workloads over time and some fraction of the client workload to the proxy cannot be satisfied by the cache. To evaluate these cases, we use synthetic read and write workloads, and do so separately because they interact with the cache in different ways.

We start with read workloads. Reads that hit in the cache achieve local performance, while reads that miss in the cache incur the full latency of accessing data in the cloud, stalling the clients accessing the data. The ratio of read hits and misses in the workload determines overall read performance, and fundamentally depends on how well the cache capacity is able to capture the file system working set across all clients in steady state.

We populate a BlueSky file system on S3 with 32 GB of data using 16 MB files.[1] We then generate a steady stream of fixed-size NFS read requests to random files through the BlueSky proxy. We vary the size of the proxy disk cache to represent different working set scenarios. In the best case, the capacity of the proxy cache is large enough to hold the entire working set: all read requests hit in the cache in steady state, minimizing latency. In the worst case, the cache capacity is zero, no part of the working set fits in the cache, and all requests go to the cloud service. In practice, a real workload falls in between these extremes. Since we make uniform random requests to any of the files, the working set is equivalent to the size of the entire file system.

Figure 7 shows that BlueSky with S3 provides good latency even when it is able to cache only 50% of the working set: with a local NFS latency of 21 ms for 32 KB requests, BlueSky is able to keep latency within 2× that value. Given that cache capacity is not an issue, this situation corresponds to clients dramatically changing the data they are accessing such that 50% of their requests are to new data objects not cached at the proxy. Larger requests take better advantage of bandwidth: 1024 KB requests are 32× larger than the 32 KB requests, but have latencies only 4× longer.

6.5 Absorbing Writes

The BlueSky proxy represents a classic write-back cache scenario in the context of a cache for a wide-area storage backend. In contrast to reads, the BlueSky proxy can absorb bursts of write traffic entirely with local performance since it implements a write-back cache. Two factors determine the proxy's ability to absorb write bursts: the capacity of the cache, which determines the instantaneous size of a burst the proxy can absorb; and the network bandwidth between the proxy and the cloud service, which determines the rate at which the proxy can drain the cache by writing back data. As long as the write workload from clients falls within these constraints, the BlueSky proxy can entirely mask the high latency to the cloud service for writes. However, if clients instantaneously burst more data than can fit in the cache, or if the steady-state write workload is higher than the bandwidth to the cloud, client writes start to experience delays that depend on the performance of the cloud service.

[1]For this and other experiments, we use relatively small file system sizes to keep the time for performing experiments manageable.

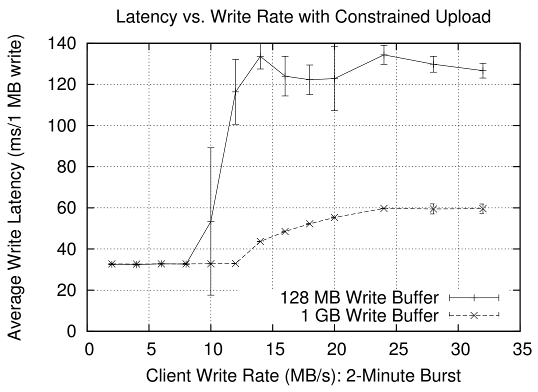


Figure 8: Write latencies when the proxy is uploading over a constrained (≈ 100 Mbps) uplink to S3 as a function of the write rate of the client and the size of the write cache to temporarily absorb writes.

We populate a BlueSky file system on S3 with 1 MB files and generate a steady stream of fixed-size 1 MB NFS write requests to random files in the file system. The client bursts writes at different rates for two minutes and then stops. So that we can overload the network between the BlueSky proxy and S3, we rate limit traffic to S3 at 100 Mbps while keeping the client↔proxy link unlimited at 1 Gbps. We start with a rate of write requests well below the traffic limit to S3, and then steadily increase the rate until the offered load is well above the limit.

Figure 8 shows the average latency of the 1 MB write requests as a function of offered load, with error bars showing standard deviation across three runs. At low write rates the latency is determined by the time to commit writes to the proxy's disk. The proxy can upload at up to about 12 MB/s to the cloud (due to the rate limiting), so beyond this point latency increases as the proxy must throttle writes by the client when the write buffer fills. With a 1 GB write-back cache the proxy can temporarily sustain write rates beyond the upload capacity. Over a 10 Mbps network (not shown), the write cache fills at correspondingly smaller client rates and latencies similarly quickly increase.

6.6 More Elaborate Workloads

Using the SPECsfs2008 benchmark we next examine the performance of BlueSky under more elaborate workload scenarios, both to subject BlueSky to more interesting workload mixes as well as to highlight the impact of different design decisions in BlueSky. We evaluate a number of different system configurations, including a native Linux nfsd in the local network (Local NFS) as well as BlueSky communicating with both Amazon S3's US-West region and Windows Azure's blob store. Unless otherwise noted, BlueSky evaluation results are for communication with Amazon S3. In addition to the base

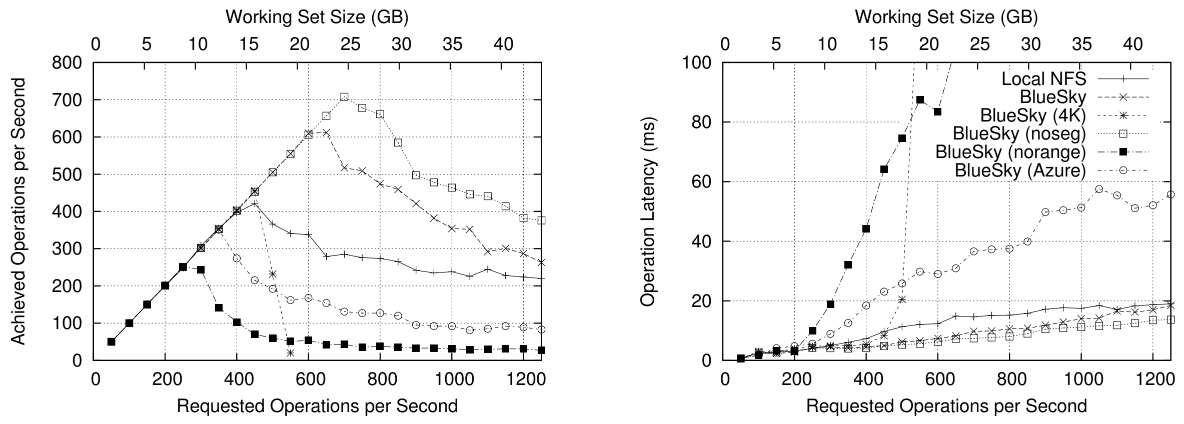


Figure 9: Comparison of various file server configurations subjected to the SPECsfs benchmark, with a low degree of parallelism (4 client processes). All BlueSky runs use cryptography, and most use Amazon US-West.

BlueSky configuration, we test a number of variants: disabling the log-structured design to store each object individually to the cloud (noseg), disabling range requests on reads so that full segments must be downloaded (norange), and using 4 KB file system blocks instead of the default 32 KB (4K). The "noseg" case is meant to allow a rough comparison with BlueSky had it been designed to store file system objects directly to the cloud (without entirely reimplementing it).

We run the SPECsfs benchmark in two different scenarios, modeling both low and high degrees of client parallelism. In the low-parallelism case, 4 client processes make requests to the server, each with at most 2 outstanding reads or writes. In the high-parallelism case, there are 16 client processes each making up to 8 reads or writes.

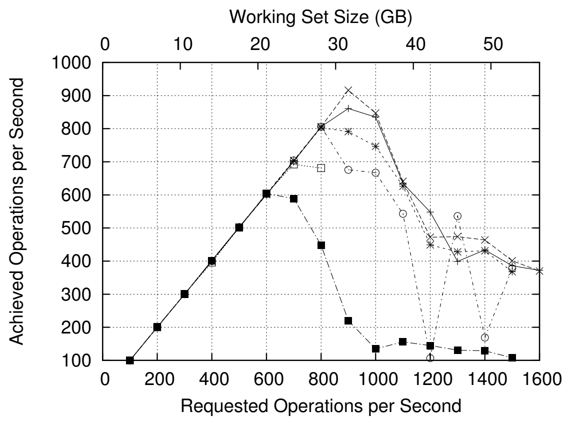
Figure 9 shows several SPECsfs runs under the low-parallelism case. In these experiments, the BlueSky proxy uses an 8 GB disk cache. The left graph shows the delivered throughput against the load offered by the load generator, and the right graph shows the corresponding average latency for the operations. At a low requested load, the file servers can easily keep up with the requests and so the achieved operations per second are equal to the requested load. As the server becomes saturated the achieved performance levels off and then decreases.

The solid curve corresponds to a local NFS server using one of the disks of the proxy machine for storage. This machine can sustain a rate of up to 420 operations/sec, at which point the disk is the performance bottleneck. The BlueSky server achieves a low latency—comparable to the local server case—at low loads since many operations hit in the proxy's cache and avoid wide-area network communication. At higher loads, performance degrades as the working set size increases. In write-heavy workloads, BlueSky incidentally performs better than the native Linux NFS server with local disk, since BlueSky commits operations to disk in a single journal and can make better use of disk bandwidth. Fundamentally, though, we consider using cloud storage successful as long as it provides performance commensurate with standard local network file systems.

BlueSky's aggregation of written data into log segments, and partial retrieval of data with byte-range requests, are important to achieving good performance and low cost with cloud storage providers. As discussed in Section 6.2, transferring data as larger objects is important for fully utilizing available bandwidth. As we show below, from a cost perspective larger objects are also better since small objects require more costly operations to store and retrieve an equal quantity of data.

In this experiment we also used Windows Azure as the cloud provider. Although Azure did not perform as well as S3, we attribute the difference primarily to the higher latency (85 ms RTT) to Azure from our proxy location (recall that we achieved equivalent maximum bandwidths to both services).

Figure 10 shows similar experiments but with a high degree of client parallelism. In these experiments, the proxy is configured with a 32 GB cache. To simulate the case in which cryptographic operations are better-accelerated, cryptography is disabled in most experiments but re-enabled in the "+crypto" experimental run. The "100 Mbps" test is identical to the base BlueSky experiment except that bandwidth to the cloud is constrained to 100 Mbps instead of 1 Gbps. Performance is comparable at first, but degrades somewhat and is more erratic under more intense workloads. Results in these experimental runs are similar to the low-parallelism case. The servers achieve a higher total throughput when there are more concurrent requests from clients. In the high-parallelism case, both BlueSky and the local NFS server provide comparable performance. Comparing cryptography enabled versus disabled, again there is very little difference: cryptographic operations are not a bottleneck.



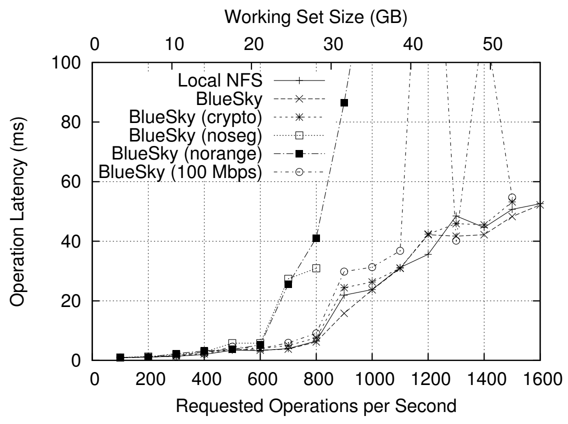


Figure 10: Comparison of various file server configurations subjected to the SPECsfs benchmark, with a high degree of parallelism (16 client processes). Most tests have cryptography disabled, but the "+crypto" test re-enables it.

	Down	*Op*	*Total*	*(Up)*
Baseline	$0.18	$0.09	$0.27	$0.56
4 KB blocks	0.09	0.07	0.16	0.47
Full segments	25.11	0.09	25.20	1.00
No segments	0.17	2.91	3.08	0.56

Table 2: Cost breakdown and comparison of various BlueSky configurations for using cloud storage. Costs are normalized to the cost per one million NFS operations in SPECsfs. Breakdowns include traffic costs for uploading data to S3 (Up), downloading data (Down), operation costs (Op), and their sum (Total). Amazon eliminated "Up" costs in mid-2011, but values using the old price are still shown for comparison.

6.7 Monetary Cost

Offloading file service to the cloud introduces monetary cost as another dimension for optimization. Figure 9 showed the relative performance of different variants of BlueSky using data from the low-parallelism SPECsfs benchmark runs. Table 2 shows the cost breakdown of each of the variants, normalized per SPECsfs operation (since the benchmark self-scales, different experiments have different numbers of operations). We use the September 2011 prices (in US Dollars) from Amazon S3 as the basis for the cost analysis: $0.14/GB stored per month, $0.12/GB transferred out, and $0.01 per 10,000 get or 1,000 put operations. S3 also offers cheaper price tiers for higher use, but we use the base prices as a worst case. Overall prices are similar for other providers.

Unlike performance, Table 2 shows that comparing by cost changes the relative ordering of the different system variants. Using 4 KB blocks had very poor performance, but using them has the lowest cost since they effectively transfer only data that clients request. The BlueSky baseline uses 32 KB blocks, requiring more data transfers and higher costs overall. If a client makes a 4 KB request, the proxy will download the full 32 KB block; many times downloading the full block will satisfy future client requests with spatial locality, but not always. Finally, the range request optimization is essential in reducing cost. When the proxy downloads an entire 4 MB segment when a client requests any data in it, the cost for downloading data increases by 150×. If providers did not support range requests, BlueSky would have to use smaller segments in its file system layout.

Although 4 KB blocks have the lowest cost, we argue that using 32 KB blocks has the best cost-performance tradeoff. The costs with 32 KB clocks are higher, but the performance of 4 KB blocks is far too low for a system that relies upon wide-area transfers

6.8 Cleaning

As with other file systems that do not overwrite in place, BlueSky must clean the file system to garbage collect overwritten data—although less to recover critical storage space, and more to save on the cost of storing unnecessary data at the cloud service. Recall that we designed the BlueSky cleaner to operate in one of two locations: running on the BlueSky proxy or on a compute instance in the cloud service. Cleaning in the cloud has compelling advantages: it is faster, does not consume proxy network bandwidth, and is cheaper since cloud services like S3 and Azure do not charge for local network traffic.

The overhead of cleaning fundamentally depends on the workload. The amount of data that needs to be read and written back depends on the rate at which existing data is overwritten and the fraction of live data in cleaned segments, and the time it takes to clean depends on both. Rather than hypothesize a range of workloads, we describe the results of a simple experiment to detail how the cleaner operates.

We populate a small BlueSky file system with 64 MB of data, split across 8 files. A client randomly writes, every few seconds, to a small portion (0.5 MB) of one of

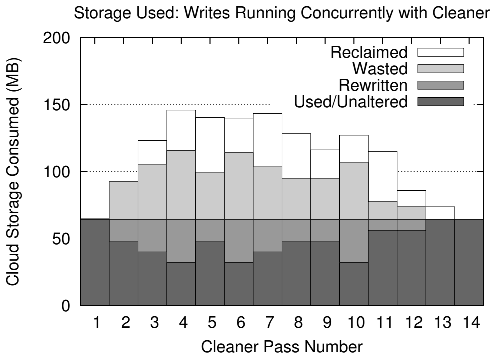


Figure 11: Storage space consumed during a write experiment running concurrently with the cleaner.

these files. Over the course of the experiment the client overwrites 64 MB of data. In parallel a cleaner runs to recover storage space and defragment file contents; the cleaner runs every 30 seconds, after the proxy incorporates changes made by the previous cleaner run. In addition to providing data about cleaner performance, this experiment validates the design that allows for safe concurrent execution of both the proxy and cleaner.

Figure 11 shows the storage consumed during this cleaner experiment; each set of stacked bars shows storage after a pass by the cleaner. At any point in time, only 64 MB of data is live in the file system, some of which (bottom dark bar) consists of data left alone by the cleaner and some of which (lighter gray bar) was rewritten by the cleaner. Some wasted space (lightest gray) cannot be immediately reclaimed; this space is either mixed useful data/garbage segments, or data whose relocation the proxy has yet to acknowledge. However, the cleaner deletes segments which it can establish the proxy no longer needs (white) to reclaim storage.

This workload causes the cleaner to write large amounts of data, because a small write to a file can cause the entire file to be rewritten to defragment the contents. Over the course of the experiment, even though the client only writes 64 MB of data the cleaner writes out an additional 224 MB of data. However, all these additional writes happen within the cloud where data transfers are free. The extra activity at the proxy, to merge updates written by the cleaner, adds only 750 KB in writes and 270 KB in reads.

Despite all the data being written out, the cleaner is able to reclaim space during experiment execution to keep the total space consumption bounded, and when the client write activity finishes at the end of the experiment the cleaner can repack the segment data to eliminate all remaining wasted space.

6.9 Client Protocols: NFS and CIFS

Finally, we use the SPECsfs benchmark to confirm that the performance of the BlueSky proxy is independent of

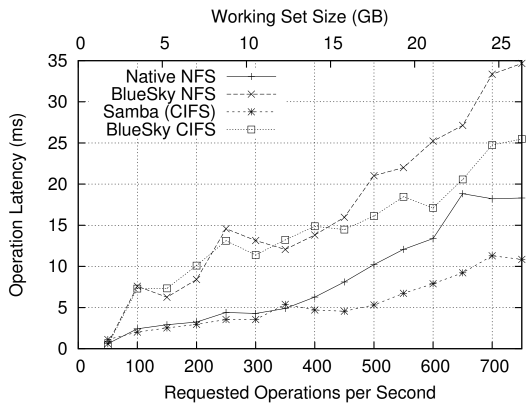


Figure 12: Latencies for read operations in SPECsfs as a function of aggregate operations per second (for all operations) and working set size.

the client protocol (NFS or CFS) that clients use. The experiments performed above use NFS for convenience, but the results hold for clients using CIFS as well.

Figure 12 shows the latency of the read operations in the benchmark as a function of aggregate operations per second (for all operations) and working set size. Because SPECsfs uses different operation mixes for its NFS and CIFS workloads, we focus on the latency of just the read operations for a common point of comparison. We show results for NFS and CIFS on the BlueSky proxy (Section 5.4) as well as standard implementations of both protocols (Linux NFS and Samba for CIFS, on which our implementation is based). For the BlueSky proxy and standard implementations, the performance of NFS and CIFS are broadly similar as the benchmark scales, and BlueSky mirrors any differences in the underlying standard implementations. Since SPECsfs uses a working set much larger than the BlueSky proxy cache capacity in this experiment, BlueSky has noticeably higher latencies than the standard implementations due to having to read data from cloud storage rather than local disk.

7 Conclusion

The promise of "the cloud" is that computation and storage will one day be seamlessly outsourced on an on-demand basis to massive data centers distributed around the globe, while individual clients will effectively become transient access portals. This model of the future (ironically similar to the old "big iron" mainframe model) may come to pass at some point, but today there are many hundreds of billions of dollars invested in the *last* disruptive computing model: client/server. Thus, in the interstitial years between now and a potential future built around cloud infrastructure, there will be a need to bridge the gap from one regime to the other.

In this paper, we have explored a solution to one such challenge: network file systems. Using a caching proxy architecture we demonstrate that LAN-oriented workstation file system clients can be transparently served by cloud-based storage services with good performance for enterprise workloads. However, we show that exploiting the benefits of this arrangement requires that design choices (even low-level choices such as storage layout) are directly and carefully informed by the pricing models exported by cloud providers (this coupling ultimately favoring a log-structured layout with in-cloud cleaning).

8 Acknowledgments

We would like to thank our shepherd Ted Wong and the anonymous reviewers for their insightful feedback, and Brian Kantor and Cindy Moore for research computing support. This work was supported in part by the UCSD Center for Networked Systems. Vrable was further supported in part by a National Science Foundation Graduate Research Fellowship.

References

[1] Amazon Web Services. Amazon Simple Storage Service. `http://aws.amazon.com/s3/`.

[2] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. DepSky: Dependable and Secure Storage in a Cloud-of-Clouds. In *EuroSys 2011*, Apr. 2011.

[3] Y. Chen and R. Sion. To Cloud Or Not To Cloud? Musings On Costs and Viability. `http://www.cs.sunysb.edu/~sion/research/cloudc2010-draft.pdf`.

[4] Cirtas. Cirtas Bluejet Cloud Storage Controllers. `http://www.cirtas.com/`.

[5] Enomaly. ElasticDrive Distributed Remote Storage System. `http://www.elasticdrive.com/`.

[6] I. Heizer, P. Leach, and D. Perry. Common Internet File System Protocol (CIFS/1.0). `http://tools.ietf.org/html/draft-heizer-cifs-v1-spec-00`.

[7] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the Winter USENIX Technical Conference*, 1994.

[8] J. Howard, M. Kazar, S. Nichols, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems (TOCS)*, 6(1):51–81, Feb. 1988.

[9] IDC. Global market pulse. `http://i.dell.com/sites/content/business/smb/sb360/en/Documents/0910-us-catalyst-2.pdf`.

[10] Jungle Disk. `http://www.jungledisk.com/`.

[11] R. Kotla, L. Alvisi, and M. Dahlin. SafeStore: A Durable and Practical Storage System. In *Proceedings of the 2007 USENIX Annual Technical Conference*, June 2007.

[12] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure Untrusted Data Repository (SUNDR). In *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2004.

[13] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud Storage with Minimal Trust. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, Oct. 2010.

[14] Microsoft. Windows Azure. `http://www.microsoft.com/windowsazure/`.

[15] Mozy. `http://mozy.com/`.

[16] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A Read/Write Peer-to-Peer File System. In *Proceedings of the 5th Conference on Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.

[17] Nasuni. Nasuni: The Gateway to Cloud Storage. `http://www.nasuni.com/`.

[18] Panzura. Panzura. `http://www.panzura.com/`.

[19] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 From Bell Labs. *USENIX Computing Systems*, 8(3):221–254, Summer 1995.

[20] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[21] Rackspace. Rackspace Cloud. `http://www.rackspacecloud.com/`.

[22] R. Rizun. s3fs: FUSE-based file system backed by Amazon S3. `http://code.google.com/p/s3fs/wiki/FuseOverAmazon`.

[23] 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.

[24] C. Ruemmler and J. Wilkes. A trace-driven analysis of disk working set sizes. Technical Report HPL-OSR-93-23, HP Labs, Apr. 1993.

[25] R. Sandberg, D. Goldberg, S. Kleirnan, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the Summer USENIX Technical Conference*, pages 119–130, 1985.

[26] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. Internet Small Computer Systems Interface (iSCSI), Apr. 2004. RFC 3720, `http://tools.ietf.org/html/rfc3720`.

[27] Standard Performance Evaluation Corporation. SPECsfs2008. `http://www.spec.org/sfs2008/`.

[28] StorSimple. StorSimple. `http://www.storsimple.com/`.

[29] TwinStrata. TwinStrata. `http://www.twinstrata.com/`.

[30] M. Vrable, S. Savage, and G. M. Voelker. Cumulus: Filesystem Backup to the Cloud. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2009.

[31] T. M. Wong and J. Wilkes. My cache or yours? Making storage more exclusive. In *Proceedings of the 2002 USENIX Annual Technical Conference*, June 2002.

[32] N. Zhu, J. Chen, and T.-C. Chiueh. TBBT: Scalable and Accurate Trace Replay for File Server Evaluation. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST)*, Dec. 2005.

Rethinking Erasure Codes for Cloud File Systems: Minimizing I/O for Recovery and Degraded Reads

Osama Khan, Randal Burns
Department of Computer Science
Johns Hopkins University

James Plank, William Pierce
Dept. of Electrical Engineering and Computer Science
University of Tennessee

Cheng Huang
Microsoft Research

Abstract

To reduce storage overhead, cloud file systems are transitioning from replication to erasure codes. This process has revealed new dimensions on which to evaluate the performance of different coding schemes: the amount of data used in recovery and when performing degraded reads. We present an algorithm that finds the optimal number of codeword symbols needed for recovery for any XOR-based erasure code and produces recovery schedules that use a minimum amount of data. We differentiate popular erasure codes based on this criterion and demonstrate that the differences improve I/O performance in practice for the large block sizes used in cloud file systems. Several cloud systems [15, 10] have adopted Reed-Solomon (RS) codes, because of their generality and their ability to tolerate larger numbers of failures. We define a new class of *rotated Reed-Solomon codes* that perform degraded reads more efficiently than all known codes, but otherwise inherit the reliability and performance properties of Reed-Solomon codes.

1 Introduction

Cloud file systems transform the requirements for erasure codes because they have properties and workloads that differ from traditional file systems and storage arrays. Our model for a cloud file system using erasure codes is inspired by Microsoft Azure [10]. It conforms well with HDFS [8] modified for RAID-6 [14] and Google's analysis of redundancy coding [15]. Some cloud file systems, such as Microsoft Azure and the Google File system, create an append-only write workload using a large block size. Writes are accumulated and buffered until a block is full and then the block is *sealed*: it is erasure coded and the coded blocks are distributed to storage nodes. Subsequent reads to sealed blocks often access smaller amounts data than the block size, depending upon workload [14, 46].

When examining erasure codes in the context of cloud file systems, two performance critical operations emerge. These are *degraded reads to temporarily unavailable data* and *recovery from single failures*. Although erasure codes tolerate multiple simultaneous failures, single failures represent 99.75% of recoveries [44]. Recovery performance has always been important. Previous work includes architecture support [13, 21] and workload optimizations for recovery [22, 48, 45]. However, it is particularly acute in the cloud owing to scale. Massive systems have frequent component failures so that recovery becomes part of regular operation [16].

Frequent and temporary data unavailability in the cloud results in degraded reads. In the period between failure and recovery, reads are *degraded* because they must reconstruct data from unavailable storage nodes using erasure codes. This is by necessity a slower operation than reading the data without reconstruction. Temporary unavailability dominates disk failures. Transient errors in which no data are lost account for more than 90% of data center failures [15], owing to network partitions, software problems, or non-disk hardware faults. For this reason, Google delays the recovery of failed storage nodes for 15 minutes. Temporary unavailability also arises systematically when software upgrades take storage nodes offline. In many data centers, software updates are a rolling, continuous process [9].

Only recently have techniques emerged to reduce the data requirements of recovering an erasure code. Two recent research projects have demonstrated how the RAID-6 codes RDP and EVENODD may recover from single disk failures by reading significantly smaller subsets of codeword symbols than the previous standard practice of recovering from the parity drive [51, 49]. Our contributions to recovery performance generalize these results to all XOR-based erasure codes, analyze existing codes to differentiate them based on recovery performance, and experimentally verify that reducing the amount of data used in recovery translates directly into improved performance for cloud file systems, but not for typical RAID array configurations.

We first present an algorithm that finds the optimal number of symbols needed for recovering data from an arbitrary number of disk failures, which also minimizes the amount of data read during recovery. We include an analysis of single failures in RAID-6 codes that reveals that sparse codes, such as Blaum-Roth [5], Liberation [34] and Liber8tion [35], have the best recovery properties, reducing data by about 30% over the standard technique that recovers each row independently. We also analyze codes that tolerate three or more disk failures, including the Reed-Solomon codes used by Google [15] and Microsoft Azure [10].

Our implementation and evaluation of this algorithm demonstrates that minimizing recovery data translates directly into improved I/O performance for cloud file systems. For large stripe sizes, experimental results track the analysis and increase recovery throughput by 30%. However, the algorithm requires the large stripes created by large sealed blocks in cloud file systems in order to amortize the seek costs incurred when reading non-contiguous symbols. This is in contrast to recovery of the smaller stripes used by RAID arrays and in traditional file systems in which the streaming recovery of all data outperforms our algorithm for stripe sizes below 1 MB. Prior work on minimizing recovery I/O [51, 49, 27] is purely analytic, whereas our work incorporates measurements of recovery performance.

We also examine the amount of data needed to perform degraded reads and reveal that it can use fewer symbols than recovery. An analysis of RAID-6 and three disk failure codes shows that degraded read performance differentiates codes that otherwise have the same recovery properties. Reads that request less than a stripe of data make the savings more acute, as much as 50%.

Reed-Solomon codes are particularly poor for degraded reads in that they must always read all data disks and parity for every degraded read. This is problematic because RS codes are popular owing to their generality and applicability to nearly all coding situations. We develop a new class of codes, *Rotated Reed-Solomon* codes, that exceed the degraded read performance of all other codes, but otherwise have the encoding performance and reliability properties of RS Codes. Rotated RS codes can be constructed for arbitrary numbers of disks and failures.

2 Related Work

Performance Metrics: Erasure codes have been evaluated historically on a variety of metrics, such as the CPU impact of encoding and decoding [3, 11, 37], the penalty of updating small amounts of data [5, 26, 52] and the ability to reconfigure systems without re-encoding [3, 7, 26]. The CPU performance of different erasure codes can vary significantly. However, for all codes that we consider, encoding and decoding bandwidth is orders of magnitude faster than disk bandwidth. Thus, the dominant factor when sealing data is writing the erasure-coded blocks to disk, not calculating the codes. Similarly, when decoding either for recovery or for degraded reads, the dominant factor is reading the data.

Updating small amounts of data is also not a concern in cloud file systems—the append-only write pattern and sealed blocks eliminate small writes in their entirety. System reconfiguration refers to changing coding parameters: changing the stripe width or increasing/decreasing fault tolerance. This type of reconfigurability is less important in clouds because each sealed block defines an independent stripe group, spread across cloud storage nodes differently than other sealed blocks. There is no single array of disks to be reconfigured. If the need for reconfiguration arises, each sealed block is re-encoded independently.

There has been some work lowering I/O costs in erasure-coded systems. In particular, WEAVER [19], Pyramid [23] and Stepped Combination Codes [18] have all been designed to lower I/O costs on recovery. However, all of these codes are non-MDS, which means that they do not have the storage efficiency that cloud storage systems demand. The REO RAID Engine [26] minimizes I/O in erasure-coded storage systems; however, its focus is primarily on the effect of updates on storage systems of smaller scale.

Cloud Storage Systems: The default storage policy in cloud file systems has become *triplication* (triple replication), implemented in the Google File system [16] and adopted by Hadoop [8] and many others. Triplication has been favored because of its ease of implementation, good read and recovery performance, and reliability.

The storage overhead of triplication is a concern, leading system designers to consider erasure coding as an alternative. The performance tradeoffs between replication and erasure coding are well understood and have been evaluated in many environments, such as peer-to-peer file systems [43, 50] and open-source coding libraries [37].

Investigations into applying RAID-6 (two fault tolerant) erasure codes in cloud file systems show that they reduce storage overheads from 200% to 25% at a small cost in reliability and the performance of large reads [14]. Microsoft research further explored the cost/benefit tradeoffs and expand the analysis to new metrics: power proportionality and complexity [53]. For these reasons, Facebook is evaluating RAID-6 and erasure codes in their cloud infrastructure [47]. Our work supports this trend, providing specific guidance as to the relative merits of different RAID-6 codes with a focus on recoverability and degraded reads.

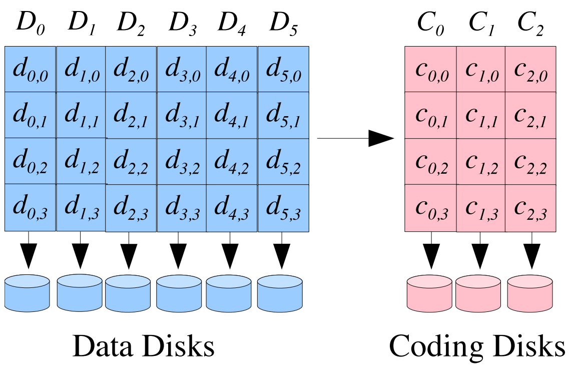
Ford et al. [15] have developed reliability models for Google's cloud file system and validated models against a year of workload and failure data from the Google infrastructure. Their analysis concludes that data placement strategies need to be aware of failure groupings and failure bursts. They also argue that, in the presence of correlated failures, codes more fault tolerant than RAID-6 are needed to to reduce exposure to data loss; they consider Reed-Solomon codes that tolerate three and four disk failures. Windows Azure storage employs Reed-Solomon codes for similar reasons [10]. The rotated RS codes that we present inherit all the properties of Reed-Solomon codes and improve degraded reads.

Recovery Optimization: Workload-based approaches to improving recovery are independent of the choice of erasure code and apply to minimum I/O recovery algorithm and rotated RS codes that we present. These include: load-balancing recovery among disks [22], recovering popular data first to decrease read degradation [48], and only recovering blocks that contain live data [45]. Similarly, architecture support for recovery can be applied to our codes, such as hardware that minimizes data copying [13] and parity declustering [21].

Reducing the amount of data used in recovery has only emerged recently as a topic and the first results have given minimum recovery schedules for EVENODD [49] and row-diagonal parity [51], both RAID-6 codes. We present an algorithm that defines the recovery I/O lower bound for any XOR-based erasure code and allows multiple codes to be compared for I/O recovery cost.

Regenerating codes provide optimal recovery bandwidth [12] among storage nodes. This concept is different than minimizing I/O; each storage node reads all of its available data and computes and sends a linear combination. Regenerating codes were designed for distributed systems in which wide-area bandwidth limits recovery performance. Exact regenerating codes [39] recover lost data exactly (not a new linear combination of data). In addition to minimizing recovery bandwidth, these codes can in some cases reduce recovery I/O. The relationship between recovery bandwidth and recovery data size remains an open problem.

RAID systems suffer reduced performance during recovery because the recovery process interferes with workload. Tian et al. [48] reorder recovery so that frequently read data are rebuilt first. This minimizes the number of reads in degraded mode. Jin et al. [25] propose reconfiguring an array from RAID-5 to RAID-0 during recovery so that reads to strips of data that are not on the failed disk do not need to be recovered. Our treatment differs in that we separate degraded reads from recovery; we make degraded reads more efficient by rebuilding just the requested data, not the entire stripe.



D_0 D_1 D_2 D_3 D_4 D_5
C_0 C_1 C_2
$d_{0,0}$ $d_{1,0}$ $d_{2,0}$ $d_{3,0}$ $d_{4,0}$ $d_{5,0}$
$d_{0,1}$ $d_{1,1}$ $d_{2,1}$ $d_{3,1}$ $d_{4,1}$ $d_{5,1}$
$d_{0,2}$ $d_{1,2}$ $d_{2,2}$ $d_{3,2}$ $d_{4,2}$ $d_{5,2}$
$d_{0,3}$ $d_{1,3}$ $d_{2,3}$ $d_{3,3}$ $d_{4,3}$ $d_{5,3}$
$c_{0,0}$ $c_{1,0}$ $c_{2,0}$
$c_{0,1}$ $c_{1,1}$ $c_{2,1}$
$c_{0,2}$ $c_{1,2}$ $c_{2,2}$
$c_{0,3}$ $c_{1,3}$ $c_{2,3}$
Data Disks
Coding Disks

Figure 1: One stripe from an erasure coded storage system. The parameters are $k = 6$, $m = 3$ and $r = 4$.

3 Background: Erasure Coded Storage

Erasure coded storage systems add redundancy for fault-tolerance. Specifically, a system of n disks is partitioned into k disks that hold data and m disks that hold coding information. The coding information is calculated from the data using an erasure code. For practical storage systems, the erasure code typically has two properties. First, it must be *Maximum Distance Separable (MDS)*, which means that if any m of the n disks fails, their contents may be recomputed from the k surviving disks. Second, it must be *systematic*, which means that the k data disks hold unencoded data.

An erasure coded storage system is partitioned into stripes, which are collections of disk blocks from each of the n disks. The blocks themselves are partitioned into *symbols*, and there is a fixed number of symbols for each disk in each stripe. We denote this quantity r. The stripes perform encoding and decoding as independent units in the disk system. Therefore, to alleviate hot spots that can occur because the coding disks may require more activity than the data disks, one can rotate the disks' identities on a stripe-by-stripe basis.

For the purpose of our analysis, we focus on a single stripe. There are k data disks labeled $D_0, \ldots, D_{k-1}$ and m coding disks labeled $C_0, \ldots, C_{m-1}$. There are nr symbols in the stripe. We label the r symbols on data disk i as $d_{i,0}, d_{i,1}, \ldots, d_{i,r-1}$ and on coding disk j as $c_{j,0}, c_{j,1}, \ldots, c_{j,r-1}$. We depict an example system in Figure 1. In this example, $k = 6$, $m = 3$ (and therefore $n = 9$) and $r = 4$.

Erasure codes are typically defined so that each symbol is a w-bit word, where w is typically small, often one. Then the coding words are defined as computations of the data words. Thus for example, suppose an erasure code were defined in Figure 1 for $w = 1$. Then each symbol in the stripe would be composed of one single bit. While that eases the definition of the erasure

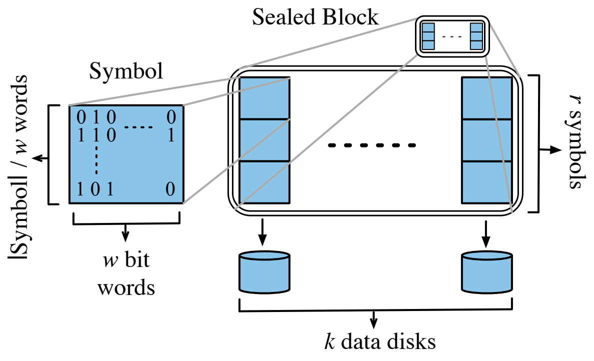


Figure 2: Relationship between words, symbols and sealed blocks.

code, it does not map directly to a disk system. In reality, it makes sense for each symbol in a sealed block to be much larger in size, on the order of kilobytes or megabytes, and for each symbol to be partitioned into w-bit words, which are encoded and decoded in parallel. Figure 2 depicts such a partitioning, where each symbol is composed of multiple words. When $w = 1$, this partitioning is especially efficient, because machines support bit operations like exclusive-or (XOR) over 64-bit and even 128-bit words, which in effect perform 64 or 128 XOR operations on 1-bit words in parallel.

When $w = 1$, the arithmetic is modulo 2: addition is XOR, and multiplication is AND. When $w > 1$, the arithmetic employed is *Galois Field* arithmetic, denoted $GF(2^w)$. In $GF(2^w)$, addition is still XOR; however multiplication is more complex, requiring a variety of implementation techniques that depend on hardware, memory, co-processing elements and w [17].

3.1 Matrix-Vector Definition

All erasure codes may be expressed in terms of a matrix-vector product. An example is pictured in Figure 3. This continues the example from Figure 1, where $k = 6$, $m = 3$ and $r = 4$; In this picture, the erasure code is defined precisely. This is a Cauchy Reed-Solomon code [6] optimized by the Jerasure library [38]. The word size, w equals one, so all symbols are treated as bits and arithmetic is composed solely of the XOR operation. The kr symbols of data are organized as a kr-element bit vector. They are multiplied by a $nr \times kr$ Generator matrix G^T.[1] The product is a vector, called the *codeword*, with nr elements. These are all of the symbols in the stripe. Each collection of r symbols in the vector is stored on a different disk in the system.

Since the the top kr rows of G^T compose an identity matrix, the first kr symbols in the codeword contain the data. The remaining mr symbols are calculated from the data using the bottom mr rows of the Generator matrix.

When up to m disks fail, the standard methodolgy for recovery is to select k surviving disks and create a decoding matrix B from the kr rows of the Generator matrix that correspond to them. The product of B^{-1} and the symbols in the k surviving disks yields the original data [6, 20, 33].

There are many MDS erasure codes that apply to storage systems. Reed-Solomon codes [40] are defined for all values of k and m. With a Reed-Solomon code, $r = 1$, and w must be such that $2^w \geq n$. Generator matrices are constructed from a Vandermonde matrix so that any $k \times k$ subset of the Generator matrix is invertible. There is quite a bit of reference material on Reed-Solomon codes as they apply to storage systems [33, 36, 6, 41], plus numerous open-source Reed-Solomon coding libraries [42, 38, 30, 31].

Cauchy Reed-Solomon codes convert Reed-Solomon codes with $r = 1$ and $w > 1$ to a code where $r = w$ and $w = 1$. In doing so, they remove the expensive multiplication of Galois Fields and replace it with additional XOR operations. There are an exponential number of ways to construct the Generator matrix of a Cauchy Reed-Solomon code. The Jerasure library attempts to construct a matrix with a minimal number of non-zero entries [38]. It is these matrices that we use in our examples with Cauchy Reed-Solomon codes.

For $m = 2$, otherwise known as RAID-6, there has been quite a bit of research on constructing codes where $w = 1$ and the CPU performance is optimized. EVENODD [3], RDP [11] and Blaum-Roth [5] codes all require $r + 1$ to be a prime number such that $k \leq r + 1$ (EVENODD) or $k \leq r$. The Liberation codes [34] require r to be a prime number and $k \leq r$, and the Liber8tion code [35] is defined for $r = 8$ and $k \leq r$. The latter three codes (Blaum-Roth, Liberation and Liber8tion) belong to a family of codes called *Minimum Density* codes, whose Generator matrices have a provably minimum number of ones.

Both EVENODD and RDP codes have been extrapolated to higher values of m [2, 4]. We call these *Generalized* EVENODD and RDP. With $m = 3$, the same restrictions on r apply. For larger values of m, there are additional restrictions on r. The STAR code [24] is an instance of the generalized EVENODD codefor $m = 3$, where recovery is performed without using the Generator matrix.

All of the above codes have a convenient feature that disk C_0 is constructed as the parity of the data disks, as in RAID-4/5. Thus, the r rows of the Generator matrix immediately below the identity portion are composed of k $(r \times r)$ identity matrices. To be consistent with these RAID systems, we will refer to disk C_0 as the "P drive."

[1]The archetypical presentation of erasure codes [26, 29, 32] typically uses the transpose of this matrix; hence, we call this matrix G^T.

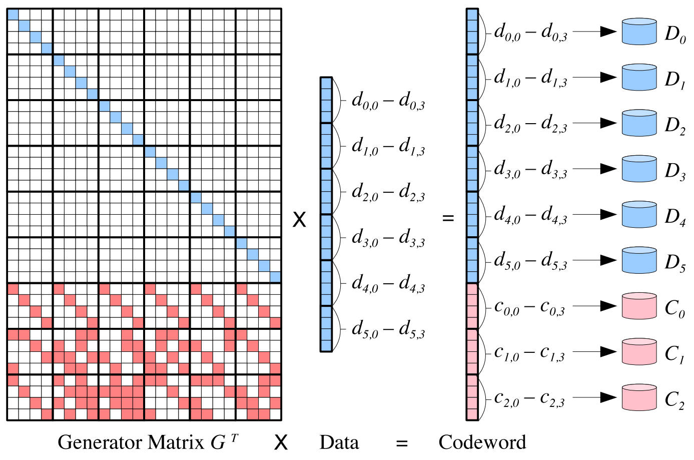


Figure 3: The matrix-vector representation of an erasure code. The parameters are the same as Figure 1: $k = 6$, $m = 3$ and $r = 4$. Symbols are one bit (i.e. $w = 1$). This is a Cauchy Reed-Solomon code for these parameters.

4 Optimal Recovery of XOR-Based Erasure codes

When a data disk fails in an erasure coded disk array, it is natural to reconstruct it simply using the P drive. Each failed symbol is equal to the XOR of corresponding symbols on each of the other data disks, and the parity symbol on the P disk. We call this methodology "Reading from the P drive." It requires k symbols to be read from disk for each decoded symbol.

Although it is straightforward both in concept and implementation, in many cases, reading from the P drive requires more I/O than is necessary. In particular, depending on the erasure code, there are savings that can be exploited when multiple symbols are recovered in the same stripe. This effect was first demonstrated by Xiang et al. in RDP systems in which one may reconstruct all the failed blocks in a stripe by reading 25 percent fewer symbols than reading from the P drive [51]. In this section, we approach the problem in general.

4.1 Algorithm to Determine the Minimum Number of Symbols for Recovery

We present an algorithm for recovering from a single disk failure in any XOR-based erasure code with a minimum number of symbols. The algorithm takes as input a Generator matrix whose symbols are single bits and the identity of a failed disk and outputs equations to decode each failed symbol. The inputs to the equations are the symbols that must be read from disk. The number of inputs is minimized.

The algorithm is computationally expensive — for the systems evaluated for this paper, each instantiation took from seconds to hours of compute-time. However, for any realistic storage system, the number of recovery scenarios is limited, so that the algorithm may be run ahead of time, and the results may be stored for when they are required by the system.

We explain the algorithm by using the erasure code of Figure 4 as an example. This small code, with $k = m = r = 2$, is not an MDS code; however its simplicity facilitates our explanation. We label the rows of G^T as R_i, $0 \leq i < nr$. Each row R_i corresponds to a data or coding symbol, and to simplify our presentation, we will refer to symbols using R_i rather than $d_{i,j}$ or $c_{i,j}$. Consider a set of symbols in the codeword whose corresponding rows in the Generator matrix sum to a vector of zeroes. One example is $\{R_0, R_2, R_4\}$. We call such a set of symbols a *decoding equation*, because the fact their rows sum to zero allows us to decode any one symbol in the set as long as the remaining symbols are not lost.

Suppose that we enumerate all decoding equations for a given Generator matrix, and suppose that some subset F of the codeword symbols are lost. For each symbol $R_i \in F$, we can determine the set E_i of decoding equations for R_i. Formally, an equation $e_i \in E_i$ if $e_i \cap F = \{R_i\}$. For example, the equation represented by the set $\{R_0, R_2, R_4\}$ may be a decoding equation in e_2 so long as neither R_0 nor R_4 is in F.

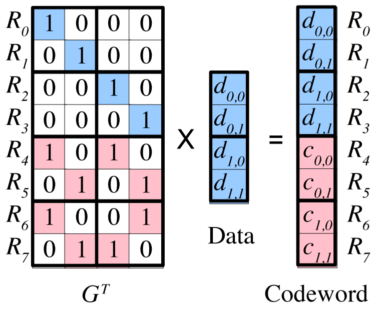


Figure 4: An example erasure code to explain the algorithm to minimize the number of symbols required to recover from failures.

We can recover all the symbols in F by selecting one decoding equation e_i from each set E_i, reading the non-failed symbols in e_i and then XOR-ing them to produce the failed symbol. To minimize the number of symbols read, our goal is to select one equation e_i from each E_i such that the number of symbols in the union of all e_i is minimized.

For example, suppose that a disk fails, and both R_0 and R_1 are lost. A standard way to decode the failed bits is to read from the P drive and use coding symbols R_4 and R_5. In equation form, $F = \{R_0, R_1\}$ $e_0 = \{R_0, R_2, R_4\}$ and $e_1 = \{R_1, R_3, R_5\}$. Since e_0 and e_1 have distinct symbols, their union is composed of six symbols, which means that four must be read for recovery. However, if we instead use $\{R_1, R_2, R_7\}$ for e_1, then $(e_0 \cup e_1)$ has five symbols, meaning that only three are required for recovery.

Thus, our problem is as follows: Given $|F|$ sets of decoding equations $E_0, E_1, \ldots E_{|F|-1}$, we wish to select one equation from each set such that the size of the union of these equations is minimized. Unfortunately, this problem is NP-Hard in $|F|$ and $|E_i|$.[2] However, we can solve the problem for practical values of $|F|$ and $|E_i|$ (typically less than 8 and 25 respectively) by converting the equations into a directed, weighted graph and finding the shortest path through the graph. Given an instance of the problem, we convert it to a graph as follows. First, we represent each decoding equation in set form as an nr-element bit string. For example, $\{R_0, R_2, R_4\}$ is represented by `10101000`.

Each node in the graph is also represented by an nr-element bit string. There is a starting node Z whose string is all zeroes. The remaining nodes are partitioned into $|F|$ sets, labeled $S_0, S_1, \ldots S_{|F|-1}$. For each equation $e_0 \in E_0$, there is a node $s_0 \in S_0$ whose bit string equals e_0's bit string. There is an edge from Z to each s_0 whose weight is equal to the number of ones in s_0's bit string.

For each node $s_i \in S_i$, there is an edge that corresponds to each $e_{i+1} \in E_{i+1}$. This edge is to a node $s_{i+1} \in S_{i+1}$ whose bit string is equal to the bitwise OR of the bit strings of s_i and e_{i+1}. The OR calculates the union of the equations leading up to s_i and e_{i+1}. The weight of the edge is equal to the difference between the number of ones in the bit strings of s_i and s_{i+1}. The shortest path from Z to any node in $S_{|F|-1}$ denotes the minimum number of elements required for recovery. If we annotate each edge with the decoding equation that creates it, then the shortest path contains the equations that are used for recovery.

To illustrate, suppose again that $F = \{R_0, R_1\}$, meaning $f_0 = R_0$ and $f_1 = R_1$. The decoding equations for E_0 and E_1 are denoted by $e_{i,j}$ where i is the index of the lost symbol in the set F and j is an index into the set E_i. E_0 and E_1 are enumerated below:

E_0	E_1
$e_{0,0}$ = `10101000`	$e_{1,0}$ = `01010100`
$e_{0,1}$ = `10010010`	$e_{1,1}$ = `01101110`
$e_{0,2}$ = `10011101`	$e_{1,2}$ = `01100001`
$e_{0,3}$ = `10100111`	$e_{1,3}$ = `01011011`

These equations may be converted to the graph depicted in Figure 5, which has two shortest paths of length five: $\{e_{0,0}, e_{1,2}\}$ and $\{e_{0,1}, e_{1,0}\}$. Both require three symbols for recovery: $\{R_2, R_4, R_7\}$ and $\{R_3, R_5, R_6\}$.

While the graph clearly contains an exponential number of nodes, one may program Dijkstra's algorithm to determine the shortest path and prune the graph drastically. For example, in Figure 5, the shortest path will be discovered before the the dotted edges and grayed nodes are considered by the algorithm. Therefore, they may be pruned.

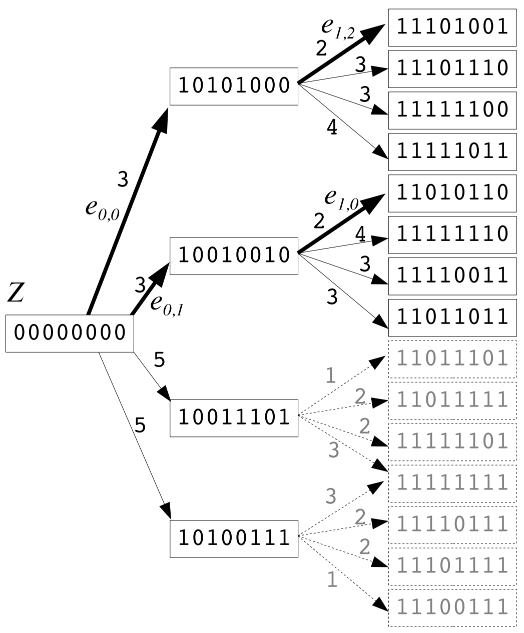


Figure 5: The graph that results when R_0 and R_1 are lost.

4.2 Algorithm for Reconstruction

When data disk i fails, the algorithm is applied for $F = \{d_{i,0}, \ldots, d_{i,r-1}\}$. When coding disk j fails, $F = \{c_{j,0}, \ldots, c_{j,r-1}\}$. If a storage system rotates the identities of the disks on a stripe-by-stripe basis, then the average number of symbols for all failed disks multiplied by the total number of stripes gives a measure of the symbols required to reconstruct a failed disk.

4.3 Algorithm for Degraded Reads

To take maximum advantage of parallel I/O, we assume that contiguous symbols in the file system are stored on

[2]Reduction from Vertex Cover.

different disks in the storage system. In other words, if one is reading three symbols starting with symbol $d_{0,0}$, then those three symbols are $d_{0,0}$, $d_{1,0}$ and $d_{2,0}$, coming from three different disk drives.

To evaluate degraded reads, we assume that an application desires to read B symbols starting at symbol $d_{x,y}$, and that data disk f has failed. We determine the penalty of the failure to be the number of symbols required to perform the read, minus B.

There are many cases that can arise from the differing values of B, f, x and y. To illustrate, first suppose that $B < k$ (which is a partial read case) and that none of the symbols to be read reside on disk f. Then the failure does not impact the read operation — it takes exactly B symbols to complete the read, and the penalty is zero.

As a second case, consider when $B = kr$ and $d_{x,y} = d_{0,0}$. Then we are reading exactly one stripe in its entirety. In this case, we have to read the $(k-1)r$ non-failed data symbols to fulfill the read request. Therefore, we may recover very easily from the P drive by reading all of its symbols and decoding. The read requires $kr = B$ symbols. Once again, the penalty is zero.

However, consider the case when $B = k$, $f = 0$, and $d_{x,y} = d_{1,0}$. Symbols $d_{1,0}$ through $d_{k-1,0}$ are non-failed and must be read. Symbol $d_{0,1}$ must also be read and it is failed. If we use the P drive to recover, then we need to read $d_{1,1}$ through $d_{k-1,0}$ and $c_{0,1}$. The total symbols read is $2k-1$: the failure has induced a penalty of $k-1$ symbols.

In all of these cases, the degraded read is contained by one stripe. If the read spans two stripes, then we can calculate the penalty as the sum of the penalties of the read in each stripe. If the read spans more than two stripes, then we only need to calculate the penalties in the first and last stripe. This is because, as described above, whole-stripe degraded reads incur no penalty.

When we perform a degraded read within a stripe, we modify our algorithm slightly. For each non-failed data symbol that must be read, we set its bit in the state of the starting node Z to one. For example, in Figure 4, suppose we are performing a degraded read where $B = 2$, $f = 0$ and $d_{x,y} = d_{0,0}$. There is one failed bit: $F = d_{0,0}$. Since $d_{1,0} = R_2$ must be read, the starting state Z of the shortest path graph is labeled `00100000`. The algorithm correctly identifies that only $c_{0,0}$ needs to be read to recover $d_{0,0}$ and complete the read.

5 Rotated Reed-Solomon Codes

Before performing analyses of failed disk reconstruction and degraded reads, we present two instances of a new erasure code, called the *Rotated Reed-Solomon* code. These codes have been designed to be MDS codes that optimize the performance of degraded reads for single disk failures. The general formulation and theoretical evaluation of these codes is beyond the scope of this paper; instead, we present instances for $m \in \{2, 3\}$.

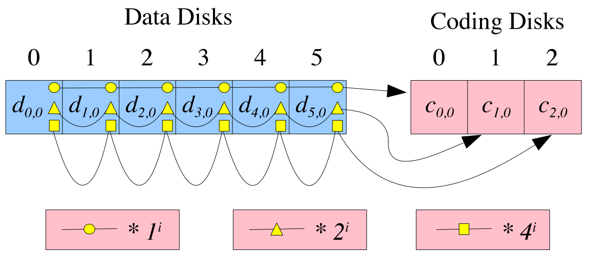


Figure 6: A Reed-Solomon code for $k = 6$ and $m = 3$. Symbols must be w-bit words such that $w \geq 4$, and arithmetic is over $GF(2^w)$.

The most intuitive way to present a Rotated Reed-Solomon code is as a modification to a standard Reed-Solomon code. We present such a code for $m \leq 3$ in Equation 1. As with all Reed-Solomon codes, $r = 1$.

$$\text{for } 0 \leq j < 3,\ c_{j,0} = \sum_{i=0}^{k-1} \left(2^j\right)^i d_{i,0} \qquad (1)$$

This is an MDS code so long as k, m, r and w adhere to some constraints, which we detail at the end of this section. This code is attractive because one may implement encoding with XOR and multiplication by two and four in $GF(2^w)$, which are all very fast operations. For example, the $m = 2$ version of this code lies at the heart of the Linux RAID-6 coding engine [1].

We present the code pictorally in Figure 6. A chain of circles denotes taking the XOR of $d_{i,0}$; a chain of triangles denotes taking the XOR of $2^i d_{i,0}$, and a chain of squares denotes taking the XOR of $4^i d_{i,0}$. To convert this code into a Rotated Reed-Solomon code, we allow r to take on any positive value, and define the coding symbols with Equation 2.

$$c_{j,b} = \sum_{i=0}^{\frac{kj}{m}-1} (2^j)^i d_{i,(b+1)\%r} + \sum_{i=\frac{kj}{m}}^{k-1} (2^j)^i d_{i,b}. \qquad (2)$$

Intuitively, the Rotated Reed-Solomon code converts the one-row code in Figure 6 into a multi-row code, and then the equations for coding disks 1 and 2 are split across adjacent rows. We draw the Rotated Reed-Solomon codes for $k = 6$ and $m = \{2, 3\}$ and $r = 3$ in Figures 7 and 8.

These codes have been designed to improve the penalty of degraded reads. Consider a RAID-6 system that performs a degraded read of four symbols starting at $d_{5,0}$ when disk 5 has failed. If we reconstruct from

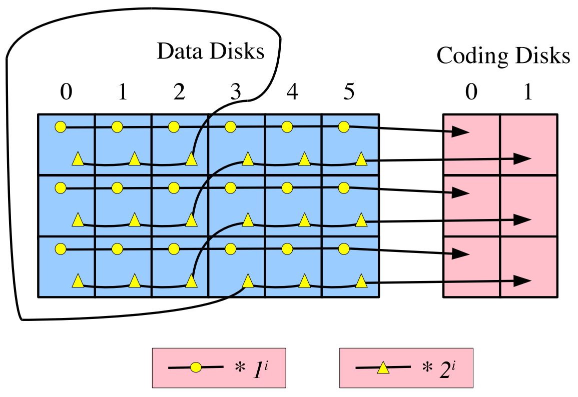


Figure 7: A Rotated Reed-Solomon code for $k = 6$, $m = 2$ and $r = 3$.

the P drive, we need to read $d_{0,0}$ through $d_{4,0}$ plus $c_{0,0}$ to reconstruct $d_{5,0}$. Then we read the non-failed symbols $d_{0,1}, d_{1,1}$ and $d_{2,1}$. The penalty is 5 symbols. With Rotated Reed-Solomon coding, $d_{5,0}$, $d_{0,1}$, $d_{1,1}$ and $d_{2,1}$ all participate in the equation for $c_{1,0}$. Therefore, by reading $c_{1,0}$, $d_{0,1}$, $d_{1,1}$, $d_{2,1}$, $d_{3,0}$ and $d_{4,0}$, one both decodes $d_{5,0}$ and reads the symbols that were required to be read. The penalty is only two symbols.

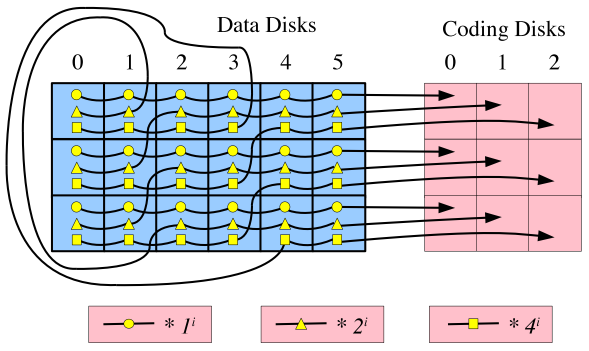


Figure 8: A Rotated Reed-Solomon code for $k = 6$, $m = 3$ and $r = 3$.

With whole disk reconstruction, when r is an even number, one can reconstruct any failed data disk by reading $\frac{r}{2}(k + \lceil \frac{k}{m} \rceil)$ symbols. The process is exemplified for $k = 6$, $m = 3$ and $r = 4$ in Figure 9. The first data disk has failed, and the symbols required to reconstruct each of the failed symbols is darkened and annotated with the equation that is used for reconstruction. Each pair of reconstructed symbols in this example shares four data symbols for reconstruction. Thus, the whole reconstruction process requires a total of 16 symbols, as opposed to 24 when reading from the P Drive.

The process is similar for the other data drives. Reconstructing failed coding drives, however does not have the same benefits. We are unaware at present of how to reconstruct a coding drive with fewer than the maximum kr symbols.

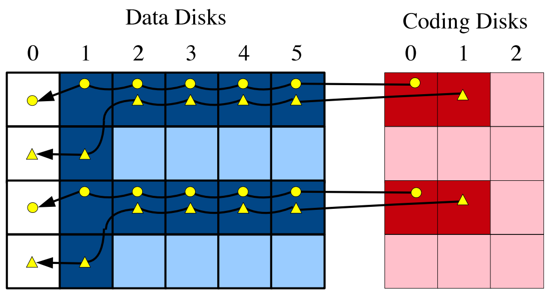


Figure 9: Reconstructing disk 0 when it fails, using Rotated Reed-Solomon coding for $k = 6$, $m = 3$, $r = 4$.

As an aside, when more than one disk fails, Rotated Reed-Solomon codes may require much more computation to recover than other codes, due to the use of matrix inversion for recovery. We view this property as less important, since multiple disk failures are rare occurrences in practical storage systems, and computational overhead is less important than the I/O impact of recovery.

5.1 MDS Constraints

The Rotated Reed-Solomon code specified above in Section 5 is not MDS in general. In other words, there are settings of k, m, w and r which cannot tolerate the failure of any m disks. Below, we detail ways to constrain these variables so that the Rotated Reed-Solomon code is MDS. Each of these settings has been verified by testing all combinations of m failures to make sure that they may be tolerated. They cover a wide variety of system sizes, certainly much larger than those in use today.

The constraints are as follows:

$$m \in \{2, 3\}$$
$$k \le 36, \text{ and } k + m \le 2^w + 1$$
$$w \in \{4, 8, 16\}$$
$$r \in \{2, 4, 8, 16, 32\}$$

Moreover, when $w = 16$, r may be any value less than or equal to 48, except 15, 30 and 45. It is a matter of future research to derive general-purpose MDS constructions of Rotated Reed-Solomon codes.

6 Analysis of Reconstruction

We evaluate the minimum number of symbols required to recover a failed disk in erasure coding systems with a variety of erasure codes. We restrict our attention to MDS codes, and systems with six data disks and either two or

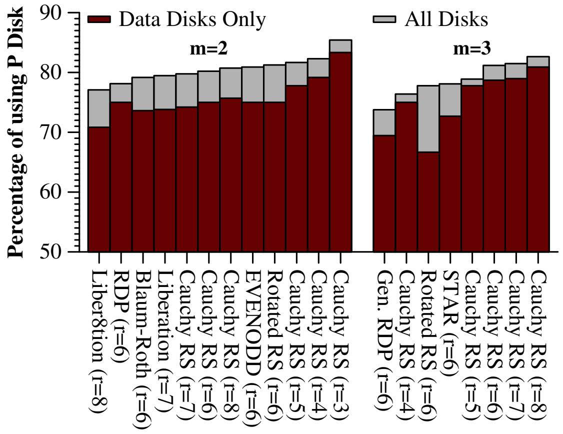


Figure 10: The minimum number of symbols required to reconstruct a failed disk in a storage system when $k = 6$ and $m \in \{2, 3\}$.

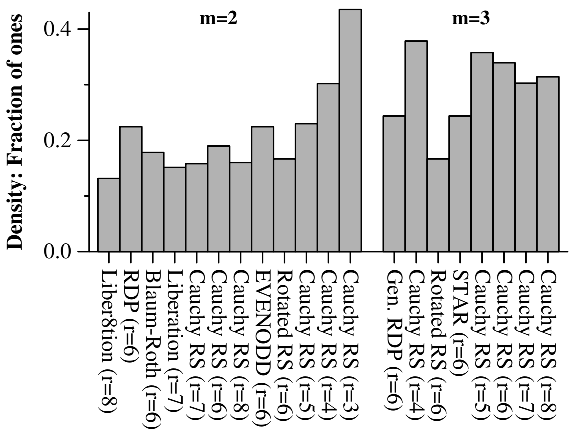


Figure 11: The density of the bottom mr rows of the Generator matrices for the codes in Figure 10.

three coding disks. We summarize the erasure codes that we test in Table 1. For each code, if r has restrictions based on k and m, we denote it in the table and include the actual values tested in the last column. All codes, with the exception of Rotated Reed-Solomon codes, are XOR codes, and all without exception define the P drive identically. Since there are a variety of Cauchy Reed-Solomon codes that can be generated for any value of k, m and r, we use the codes generated by the Jerasure coding library, which attempts to minimize the number of non-zero bits in the Generator matrix [38].

Code	m	Restrictions on r	r tested
EVENODD [3]	2	$r+1$ prime $\geq k$	6
RDP [11]	2	$r+1$ prime $> k$	6
Blaum-Roth [5]	2	$r+1$ prime $> k$	6
Liberation [34]	2	r prime $\geq k$	7
Liber8tion [35]	2	$r = 8, r \geq k$	8
STAR [24]	3	$r+1$ prime $\geq k$	6
Generalized RDP [2]	3	$r+1$ prime $> k$	6
Cauchy RS [6]	2,3	$2^r \geq n$	3-8
Rotated	2,3	None	6

Table 1: The erasure codes and values of r tested.

For each code listed in Table 1, we ran the algorithm from section 4.1 to determine the minimum number of symbols required to reconstruct each of the $k+m$ failed disks in one stripe. The average number is plotted in Figure 10. The Y-axis of these graphs are expressed as a percentage of kr, which represents the number of symbols required to reconstruct from the P drive. This is also the number of symbols required when standard Reed-Solomon coding is employed.

In both sides of the figure, the codes are ordered from best to worst, and two bars are plotted: the average number of symbols required when the failed disk is a data disk, and when the failed disk can be either data or coding. In all codes, the performance of decoding data disks is better than re-encoding coding disks. As mentioned in Section 5, Rotated Reed-Solomon codes require kr symbols to re-encode. In fact, the C_1 drive in *all* the RAID-6 codes require kr symbols to re-encode. Regardless, we believe that presenting the performance for data and coding disks is more pertinent, since disk identities are likely to be rotated from stripe to stripe, and therefore a disk failure will encompass all n decoding scenarios.

For the RAID-6 systems, the minimum density codes (Blaum-Roth, Liberation and Liber8tion) as a whole exhibit excellent performance, especially when data disks fail. It is interesting that the Liber8tion code, whose construction was the result of a theory-less enumeration of matrices, exhibits the best performance.

Faced with these results, we sought to determine if Generator matrix density has a direct impact on disk recovery. Figure 11 plots the density of the bottom mr rows of the Generator matrices for each of these codes. To a rough degree, density is correlated to recovery performance of the data disks; however the correlation is only approximate. The precise relationship between codes and their recovery performance is a direction of further research.

Regardless, we do draw some important conclusions from the work. The most significant one is that reading from the P drive or using standard Reed-Solomon codes is not a good idea in cloud storage systems. If recovery performance is a dominant concern, then the Liber8tion code is the best for RAID-6, and Generalized RDP is the best for three fault-tolerant systems.

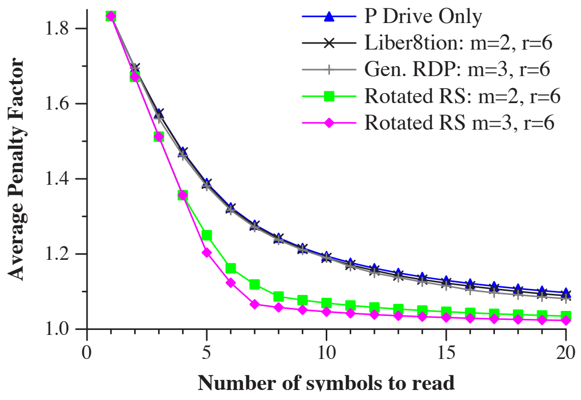


Figure 12: The penalty of degraded reads in storage systems with $k = 6$.

7 Analysis of Degraded Reads

To evaluate degraded reads, we compute the average penalty of degraded reads for each value of B from one to 20. The average is over all k potential data disks failing and over all kr potential starting points for the read (all potential $d_{x,y}$). This penalty is plotted in Figure 12 as a factor of B, so that the impact of the penalty relative to the size of the read is highlighted. Since whole-stripe reads incur no penalty, the penalty of all values of $B \geq kr$ are the same, which means that as B grows, the penalty factor approaches one. Put another way, large degraded reads incur very little penalty.

We plot only a few erasure codes because, with the exception of Rotated Reed-Solomon codes, all perform roughly the same. The Rotated Reed-Solomon codes, which were designed expressly for degraded reads, require significantly fewer symbols on degraded reads. This is most pronounced when B is between 5 and 10. To put the results in context, suppose that symbols are 1 MB and that a cloud application is reading collections of 10 MB files such as MP3 files. If the system is in degraded mode, then using Rotated Reed-Solomon codes with $m = 3$ incurs a penalty of 4.6%, as opposed to 19.6% using regular Reed-Solomon codes.

Combined with their good performance with whole-disk recovery, the Rotated Reed-Solomon codes provide a very good blend of properties for cloud storage systems. Compared to regular Reed-Solomon codes, or to recovery strategies that employ only the P-drive for single-disk failures, their improvement is significant.

8 Evaluation

We have built a storage system to evaluate the recovery of sealed blocks. The goal of our experiments is to determine the points at which the theoretical results of sections 6 and 7 apply to storage systems configured as cloud file system nodes.

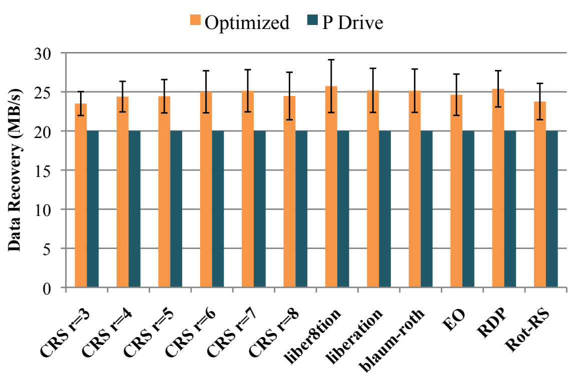


Figure 13: The I/O performance of RAID-6 codes recovering from a single disk failure averaged over all disks (data and parity).

Experimental Setup: All experiments are run on a 12-disk array using a SATA interface running on a quad-core Intel Xeon E5620 processor with 2GB of RAM. Each disk is a Seagate ST3500413AS Barracuda with 500 GB capacity and operates at 7200 rpm. The Jerasure v1.2 library was used for construction and manipulation of the Generator matrices and for Galois Field arithmetic in rotated Reed-Solomon coding [38]. All tests mirror the configurations in Table 1, evaluating a variety of erasure codes for which $k = 6$ and $m \in \{2, 3\}$. Each data point is the average of twenty trials. Error bars denote a standard deviation from the mean.

Evaluating Reconstruction: In these experiments, we affix the symbol size at 16 MB, which results in sealed blocks containing between 288 and 768 MB, depending on the values of r and k. After creating a sealed block, we measure the performance of reconstructing each of the $k + m$ disks when it fails. We plot the average performance in Figures 13 and 14. Each erasure code contains two measurements: the performance of recovering from the P drive, and the performance of optimal recovery. The data recovery rate is plotted. This is the speed of recovering the lost symbols of data from the failed disk.

As demonstrated in Figure 13, for the RAID-6 codes, optimal recovery improves performance by a factor of 15% to 30%, with Minimum-Density codes realizing the largest performance gains. As the analysis predicts, the Liber8tion code outperforms all other codes. In general, codes with large r and less density have better performance. Cauchy Reed-Solomon codes with $r \geq 6$ compare well, but with $r = 3$, they give up about 10% of recovery performance. The rotated RS code performs roughly the same as Cauchy-RS codes with $r = 8$.

Figure 14 confirms that Generalized-RDP substantially outperforms the other codes. Cauchy Reed-Solomon codes have different structure for $m = 3$ than

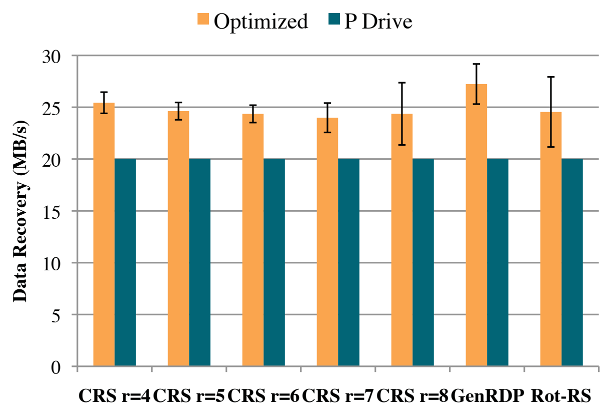


Figure 14: The I/O performance of $m = 3$ codes recovering from a single disk failure.

$m = 2$, with smaller r offering better performance. This result matches the analysis in Section 6, but is surprising nonetheless, because the smaller r codes are denser.

The large block size in cloud file systems means that data transfer dominates recovery costs. All of the codes read data at about 120 MB/s on aggregate. The results in Figures 13 and 14 match those in Figure 10 closely. We explore the effect of the symbol size and, thus, the sealed block size in the next experiment.

Size of Sealed Blocks: Examining the relationship between recovery performance and the amount of the data underlying each symbol shows that optimal recovery works effectively only for relatively large sealed blocks. Figure 15 plots the recovery data rate as a function of symbol size for GenRDP and Liber8tion with and without optimal recovery. We chose these codes because their optimized version uses the fewest recovery symbols at $m = 2$ (Liber8tion) and $m = 3$ (GenRDP). Our disk array recovers data sequentially at approximately 20 MB/s. This rate is realized for erasure codes with any value of r when the code is laid out on an array of disks. Recovery reads each disk in a sequential pass and rebuilds the data. Unoptimized GenRDP and Liber8tion approach this rate with increasing symbol size. Full sequential performance is realized for symbols of size 16M or more, corresponding to sealed blocks of size 768 MB for Liber8tion and 576 MB for GenRDP.

We parameterize experiments by symbol size because recovery performance scales with the symbol size. Optimal recovery determines the minimum number of symbols needed and accesses each symbol independently, incurring a seek penalty for most symbols: those not adjacent to other needed symbols. For small symbols, this recovery process is inefficient. There is some noise in our data at for symbols of size 64K and 256K that comes from disk track read-ahead and caching.

Optimal recovery performance exceeds the stream-

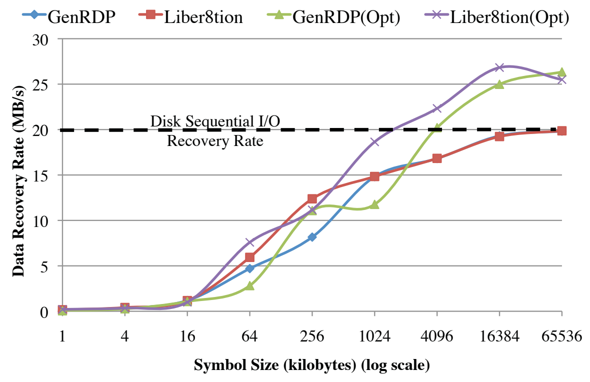


Figure 15: Data recovery rate as a function of the codeword symbol size.

ing recovery rate above 4M symbols, converging to the throughput expected by analysis as disk seeks become fully amortized. Sealed blocks using these parameters can expect the recovery performance of distributed erasure codes to exceed that realized by disk arrays.

As symbols and stripes become too large, recovery requires more memory than is available and performance degrades. The 64 MB point for Liber8tion(Opt) with $r = 8$ shows a small decline from 16 MB, because the encoded stripe is 2.4 GB, larger than the 2G of memory on our system.

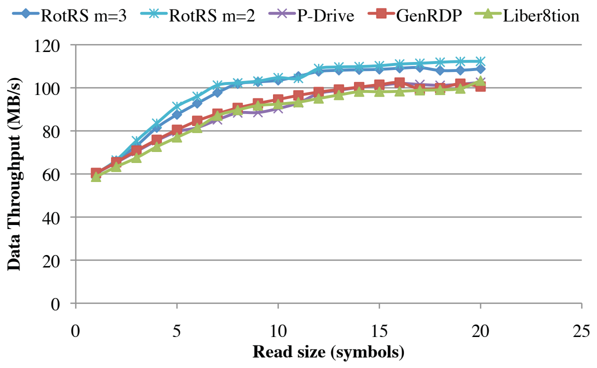


Figure 16: The throughput of degraded reads as a function of the number of symbols read.

Degraded Reads: Figure 16 plots the performance of degraded reads as a function of the number of symbols read with $k = 6$ and 16 MB per symbol. We compare Rotated Reed-Solomon codes with P Drive recovery and with the best performing optimal recovery codes, Liber8tion for $m = 2$ and GenRDP for $m = 3$. We measure the degraded read performance of read requests ranging from 1 symbol to 20 symbols. For each read size, we measure the performance of starting at each of the potential kr starting blocks in the stripe, and plot the average speed of the read when each data disk fails. The results match Figure 12 extremely closely. When reading one symbol, all algorithms perform identically, because

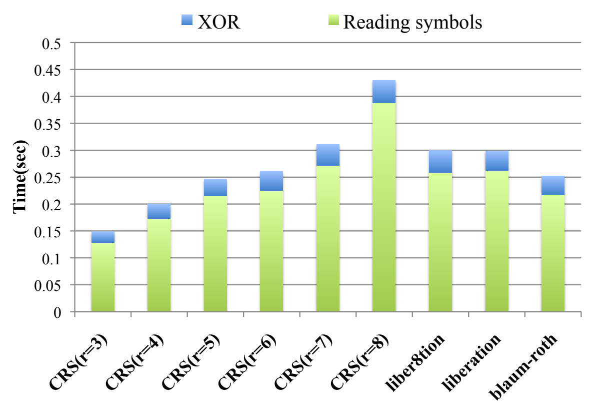


Figure 17: Relative cost of computation of XORs and read I/O during recovery.

they all either read the symbol from a non-failed disk or they must read six disks to reconstruct. When reading eight symbols, Rotated Reed-Solomon coding shows the most improvement over the others, reading 13% faster than Liber8tion ($m = 2$) and Generalized RDP ($m = 3$). As predicted by Figure 12, the improvement lessens as the number of symbols read increases. The overall speed of all algorithms improves as the number of symbols read increases, because fewer data blocks are read for recovery and then thrown away.

The Dominance of I/O: We put forth that erasure codes should be evaluated based on the the data used in recovery and degraded reads. Implicit in this thesis is that the computation for recovery is inconsequential to overall performance. Figure 17 shows the relative I/O costs and processing time for recovery of a single disk failure. A single stripe with a 1 MB symbol was recovered for each code. Codes have different stripe sizes. Computation cost never exceeds 10% of overall costs. Furthermore, this computation can be overlapped with I/O when recovering multiple sealed blocks.

9 Discussion

Our findings provide guidance as to how to deploy erasure coding in the cloud file systems with respect to choosing a specific code and the size of sealed blocks. Cloud file systems distribute the coded blocks from each stripe (sealed block) on a different set of storage nodes. This strategy provides load balance and incremental scalability in the data center. It also prevents correlated failures from resulting in data loss and mitigates the effect that any single failure has on a data set or application [15]. However, it does mean that each stripe is recovered independently from a different set of disks. To achieve good recovery performance when recovering independent stripes, codeword symbols need to be large enough to amortize disk seek overhead. Our results recommend a minimum symbol size of 4 MB and prefer 16 MB. This translates to a minimum sealed block size of 144 MB and preferred size of 576 MB for RDP and GenRDP, for example. Cloud file systems would benefit from increasing the sealed blocks to these size from the 64 MB default. Increasing the symbol size has drawbacks as well. It increases memory consumption during recovery and increases the latency of degraded reads, because larger symbols need to recover more data.

Codes differ substantially in recovery performance, which demands a careful selection of code and parameters for cloud file systems. Optimally-sparse, Minimum-Density codes tend to perform best. The Liber8tion code and Generalized RDP are preferred for $m = 2$ and $m = 3$ respectiveley. Reed-Solomon codes will continue to be popular for their generality. For some Reed-Solomon codes, including rotated-RS codes, recovery performance may be improved by more than 20%. However, the number of symbols per disk (r) has significant impact. For $k = 6$ data disks, the best values are $r = 7$ for $m = 2$ and $r = 4$ for $m = 3$.

Several open problems remain with respect to optimal recovery and degraded reads. While our algorithm can determine the minimum number of symbols needed for recovery for any given code, it remains unknown how to generate recovery-optimal erasure codes. We are pursuing this problem both analytically and through a programatic search of feasible generator matrixes. Rotated RS codes are a first result in lowering degraded read costs. Lower bounds for the number of symbols needed for degraded reads have not been determined.

We have restricted our treatment to MDS codes, since they are used almost exclusively in practice because of their optimal storage efficiency. However, some codes with decreased storage efficiency have much lower recovery costs than MDS [27, 18, 28, 23, 19]. Exploring non-MDS codes more thoroughly will help guide those building cloud systems in the tradeoffs between storage efficiency, fault-tolerance, and performance.

Acknowledgments

We thank Sayeed Choudhury, Timothy DiLauro and other members of the Data Conservancy Project who provided valuable guidance on the requirements of data availablity in preservation enviroments. We also thank Reza Curtmola, Ragib Hasan, John Griffin, Guiseppe Ateniese and our peers at the Johns Hopkins Storage Systems Lab for their technical input. This work was supported by the National Science Foundation awards CSR-1016636, CCF-0937810, OCI-0830876, and DUE-0734862 as well as by NetApp and Microsoft Research.

References

[1] H. P. Anvin. The mathematics of RAID-6. `http://kernel.org/pub/linux/kernel/people/hpa/raid6.pdf`, 2009.

[2] M. Blaum. A family of MDS array codes with minimal number of encoding operations. In *IEEE International Symposium on Information Theory*, September 2006.

[3] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures. *IEEE Transactions on Computing*, 44(2):192– 202, February 1995.

[4] M. Blaum, J. Bruck, and A. Vardy. MDS array codes with independent parity symbols. *IEEE Transactions on Information Theory*, 42(2):529–542, February 1996.

[5] M. Blaum and R. M. Roth. On lowest density MDS codes. *IEEE Transactions on Information Theory*, 45(1):46–59, January 1999.

[6] J. Blomer, M. Kalfane, M. Karpinski, R. Karp, M. Luby, and D. Zuckerman. An XOR-based erasure-resilient coding scheme. Technical Report TR-95-048, International Computer Science Institute, August 1995.

[7] V. Bohossian and J. Bruck. Shortening array codes and the perfect 1-Factorization conjecture. In *IEEE International Symposium on Information Theory*, pages 2799–2803, 2006.

[8] D. Borthakur. The Hadoop distributed file system: Architecture and design. `http://hadoop.apache.org/common/docs/current/hdfs-design.html`, 2009.

[9] E. Brewer. Lessons from giant-scale services. *Internet Computing*, 5(4), 2001.

[10] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. Fahim ul Haq, M. Ikram ul Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure storage: A highly available cloud storage service with strong consistency. In *Symposium on Operating Systems Principles*, 2011.

[11] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row diagonal parity for double disk failure correction. In *Conference on File and Storage Technologies*, March 2004.

[12] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran. Network coding for distributed storage systems. *IEEE Trans. Inf. Theor.*, 56(9):4539–4551, September 2010.

[13] A. L. Drapeau et al. RAID-II: A high-bandwidth network file server. In *International Symposium on Computer Architecture*, 1994.

[14] B. Fan, W Tanisiriroj, L. Xiao, and G. Gibson. DiskReduce: RAID for data-intensive scalable computing. In *Parallel Data Storage Workshop*, 2008.

[15] D. Ford, F. Labelle, F. .I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed file systems. In *Operating Systems Design and Implementation*, 2010.

[16] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. In *ACM SOSP*, 2003.

[17] K. Greenan, E. Miller, and T. J. Schwartz. Optimizing Galois Field arithmetic for diverse processor architectures and applications. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, September 2008.

[18] K. M. Greenan, X. Li, and J. J. Wylie. Flat XOR-based erasure codes in storage systems: Constructions, efficient recovery, and tradeoffs. *Mass Storage Systems and Technologies*, 2010.

[19] J. L. Hafner. Weaver codes: Highly fault tolerant erasure codes for storage systems. In *Conference on File and Storage Technologies*, 2005.

[20] J. L. Hafner, V. Deenadhayalan, K. K. Rao, and J. A. Tomlin. Matrix methods for lost data reconstruction in erasure codes. In *Conference on File and Storage Technologies*, 2005.

[21] M. Holland and G. A. Gibson. Parity declustering for continuous operation in redundant disk arrays. In *Architectural Support for Programming Languages and Operating Systems*. ACM, 1992.

[22] R. Y. Hou, J. Menon, and Y. N. Patt. Balancing I/O response time and disk rebuild time in a RAID5 disk array. In *Hawai'i International Conference on System Sciences*, 1993.

[23] C. Huang, M. Chen, and J. Li. Pyramid codes: Flexible schemes to trade space for access efficiency in reliable data storage systems. *Network Computing and Applications*, 2007.

[24] C. Huang and L. Xu. STAR: An efficient coding scheme for correcting triple storage node failures. *IEEE Transactions on Computers*, 57(7):889–901, July 2008.

[25] H. Jin, J. Zhang, and K. Hwang. A raid reconfiguration scheme for gracefully degraded operations. *EuroMicro Conference on Parallel, Distributed, and Network-Based Processing*, 0:66, 1999.

[26] D. Kenchammana-Hosekote, D. He, and J. L. Hafner. REO: A generic RAID engine and optimizer. In *Conference on File and Storage Technologies*, pages 261–276, 2007.

[27] O. Khan, R. Burns, J. S. Plank, and C. Huang. In search of I/O-optimal recovery from disk failures. In *Workshop on Hot Topics in Storage Systems*, 2011.

[28] M. Luby, M. Mitzenmacher, A. Shokrollahi, D. Spielman, and V. Stemann. Practical loss-resilient codes. In *29th Annual ACM Symposium on Theory of Computing*, pages 150–159, El Paso, TX, 1997. ACM.

[29] F. J. MacWilliams and N. J. A. Sloane. *The Theory of Error-Correcting Codes, Part I*. North-Holland Publishing Company, 1977.

[30] Onion Networks. Java FEC Library v1.0.3. Open source code distribution: `http://onionnetworks.com/fec/javadoc/`, 2001.

[31] A. Partow. Schifra Reed-Solomon ECC Library. Open source code distribution: `http://www.schifra.com/downloads.html`, 2000-2007.

[32] W. W. Peterson and E. J. Weldon, Jr. *Error-Correcting Codes, Second Edition*. The MIT Press, 1972.

[33] J. S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software—Practice & Experience*, 27(9):995–1012, 1997.

[34] J. S. Plank. The RAID-6 Liberation codes. In *Conference on File and Storage Technologies*, 2008.

[35] J. S. Plank. The RAID-6 Liber8Tion code. *Int. J. High Perform. Comput. Appl.*, 23:242–251, August 2009.

[36] J. S. Plank and Y. Ding. Note: Correction to the 1997 tutorial on Reed-Solomon coding. *Software – Practice & Experience*, 35(2):189–194, February 2005.

[37] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. Wilcox-OHearn. A performance evaluation and examination of open-source erasure coding libraries for storage. In *Conference on File and Storage Technologies*, 2009.

[38] J. S. Plank, S. Simmerman, and C. D. Schuman. Jerasure: A library in C/C++ facilitating erasure coding for storage applications - Version 1.2. Technical Report CS-08-627, University of Tennessee, August 2008.

[39] K. V. Rashmi, N. B. Shah, P. V. Kumar, and K. Ramachandran. Explicit construction of optimal exact regenerating codes for distributed storage. In *Communication, Control, and Computing*, 2009.

[40] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8:300–304, 1960.

[41] L. Rizzo. Effective erasure codes for reliable computer communication protocols. *ACM SIGCOMM Computer Communication Review*, 27(2):24–36, 1997.

[42] L. Rizzo. Erasure codes based on Vandermonde matrices. Gzipped **tar** file posted at `http://planete-bcast.inrialpes.fr/rubrique.php3?id_rubrique=10`, 1998.

[43] R. Rodrigues and B. Liskov. High availability in DHTS: Erasure coding vs. replication. In *Workshop on Peer-to-Peer Systems*, 2005.

[44] B. Schroeder and G. Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 mean to you? In *Conference on File and Storage Technologies*, 2007.

[45] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving storage system availability with D-GRAID. In *Conference on File and Storage Technologies*, 2004.

[46] Apache Software. Pigmix. `https://cwiki.apache.org/confluence/display/PIG/PigMix`, 2011.

[47] A. Thusoo, D. Borthakur, R. Murthy, Z. Shao, N. Jain, H. Liu, S. Anthony, and J. S. Sarma. Data warehousing and analytics infrastructure at Facebook. In *SIGMOD*, 2010.

[48] L. Tian, D. Feng, H. Jiang, K. Zhou, L. Zeng, J. Chen, Z. Wang, and Z. Song. PRO: a popularity-based multi-threaded reconstruction optimization for RAID-structured storage systems. In *Conference on File and Storage Technologies*, 2007.

[49] Z. Wang, A. G. Dimakis, and J. Bruck. Rebuilding for array codes in distributed storage systems. *CoRR*, abs/1009.3291, 2010.

[50] H. Weatherspoon and J. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Workshop on Peer-to-Peer Systems*, 2002.

[51] L. Xiang, Y. Xu, J. C. S. Lui, and Q. Chang. Optimal recovery of single disk failure in RDP code storage systems. In *ACM SIGMETRICS*, 2010.

[52] L. Xu and J. Bruck. X-Code: MDS array codes with optimal encoding. *IEEE Transactions on Information Theory*, 45(1):272–276, January 1999.

[53] Z. Zhang, A. Deshpande, X. Ma, E. Thereska, and D. Narayanan. Does erasure coding have a role to play in my data center? Microsoft Technical Report MSR-TR-2010-52, 2010.

NCCloud: Applying Network Coding for the Storage Repair in a Cloud-of-Clouds

Yuchong Hu[†], Henry C. H. Chen[†], Patrick P. C. Lee[†], Yang Tang[‡]
[†]*The Chinese University of Hong Kong*, [‡]*Columbia University*
ychu@inc.cuhk.edu.hk, {chchen,pclee}@cse.cuhk.edu.hk, ty@cs.columbia.edu

Abstract

To provide fault tolerance for cloud storage, recent studies propose to stripe data across multiple cloud vendors. However, if a cloud suffers from a permanent failure and loses all its data, then we need to repair the lost data from other surviving clouds to preserve data redundancy. We present a proxy-based system for multiple-cloud storage called NCCloud, which aims to achieve cost-effective repair for a permanent single-cloud failure. NCCloud is built on top of network-coding-based storage schemes called regenerating codes. Specifically, we propose an implementable design for the functional minimum-storage regenerating code (F-MSR), which maintains the same data redundancy level and same storage requirement as in traditional erasure codes (e.g., RAID-6), but uses less repair traffic. We implement a proof-of-concept prototype of NCCloud and deploy it atop local and commercial clouds. We validate the cost effectiveness of F-MSR in storage repair over RAID-6, and show that both schemes have comparable response time performance in normal cloud storage operations.

1 Introduction

Cloud storage provides an on-demand remote backup solution. However, using a single cloud storage vendor raises concerns such as having a single point of failure [3] and vendor lock-ins [1]. As suggested in [1, 3], a plausible solution is to stripe data across different cloud vendors. While striping data with conventional erasure codes performs well when some clouds experience *short-term* failures or *foreseeable* permanent failures [1], there are real-life cases showing that permanent failures do occur and are not always foreseeable [23, 14, 20].

This work focuses on *unexpected* cloud failures. When a cloud fails permanently, it is important to activate *storage repair* to maintain the level of data redundancy. A repair operation reads data from existing surviving clouds and reconstructs the lost data in a new cloud. It is desirable to reduce the repair traffic, and hence the monetary cost, due to data migration.

Recent studies (e.g., [6, 8, 16, 25]) propose *regenerating codes* for distributed storage. Regenerating codes are built on the concept of network coding [2]. They aim to intelligently mix data blocks that are stored in existing storage nodes, and then regenerate data at a new storage node. It is shown that regenerating codes reduce the data repair traffic over traditional erasure codes subject to the same fault-tolerance level. Despite the favorable property, regenerating codes are mainly studied in the theoretical context. It remains uncertain regarding the practical performance of regenerating codes, especially with the encoding overhead incurred in regenerating codes.

In this paper, we propose *NCCloud*, a proxy-based system designed for multiple-cloud storage. We propose the *first* implementable design for the *functional minimum-storage regenerating code (F-MSR)* [8], and in particular, we eliminate the need of performing encoding operations within storage nodes as in existing theoretical studies. Our F-MSR implementation maintains double-fault tolerance and has the same storage cost as in traditional erasure coding schemes based on RAID-6, but uses less repair traffic when recovering a single-cloud failure. On the other hand, unlike most erasure coding schemes that are *systematic* (i.e., original data chunks are kept), F-MSR is *non-systematic* and stores only linearly combined code chunks. Nevertheless, F-MSR is suited to rarely-read long-term archival applications [6].

We show that in a practical deployment setting, F-MSR can save the repair cost by 25% compared to RAID-6 for a four-cloud setting, and up to 50% as the number of clouds further increases. In addition, we conduct extensive evaluations on both local cloud and commercial cloud settings. We show that our F-MSR implementation only adds a small encoding overhead, which can be easily masked by the file transfer time over the Internet. Thus, our work validates the practicality of F-MSR via NCCloud, and motivates further studies of realizing regenerating codes in large-scale deployments.

2 Motivation of F-MSR

We consider a distributed, multiple-cloud storage setting from a client's perspective, such that we stripe data over multiple cloud vendors. We propose a proxy-based design [1] that interconnects multiple cloud repositories, as shown in Figure 1(a). The proxy serves as an interface between client applications and the clouds. If a cloud experiences a permanent failure, the proxy activates the repair operation, as shown in Figure 1(b). That is, the

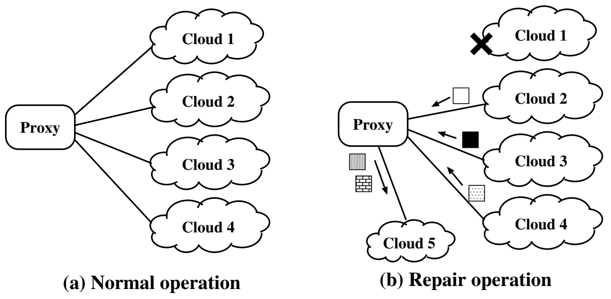


Figure 1: Proxy-based design for multiple-cloud storage: (a) normal operation, and (b) repair operation when Cloud node 1 fails. During repair, the proxy regenerates data for the new cloud.

proxy reads the essential data pieces from other surviving clouds, reconstructs new data pieces, and writes these new pieces to a new cloud. Note that this repair operation does not involve direct interactions among the clouds.

We consider fault-tolerant storage based on *maximum distance separable (MDS)* codes. Given a file object, we divide it into equal-size *native chunks*, which in a non-coded system, would be stored on k clouds. With coding, the native chunks are encoded by linear combinations to form *code chunks*. The native and code chunks are distributed over $n > k$ clouds. When an MDS code is used, the original file object may be reconstructed from the chunks contained in *any* k of the n clouds. Thus, it tolerates the failure of any $n - k$ clouds. We call this feature the *MDS property*. The extra feature of F-MSR is that reconstructing a single native or code chunk may be achieved by reading up to 50% less data from the surviving clouds than reconstructing the whole file.

This paper considers a multiple-cloud setting that is double-fault tolerant (e.g., RAID-6) and provides data availability toward at most two cloud failures (e.g., a few days of outages [7]). That is, we set $k = n - 2$. We expect that such a fault tolerance level suffices in practice. Given that a permanent failure is less frequent but possible, our primary objective is to minimize the cost of storage repair for a permanent single-cloud failure, due to the migration of data over the clouds.

We define the *repair traffic* as the amount of outbound data being read from other surviving clouds during the single-cloud failure recovery. Our goal is to minimize the repair traffic for cost-effective repair. Here, we do not consider the inbound traffic (i.e., the data being written to a cloud), as it is free of charge in many cloud vendors (see Table 1 in Section 5).

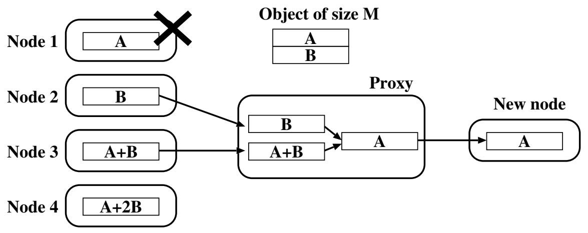
We now show how F-MSR saves the repair traffic via an example. Suppose that we store a file of size M on four clouds, each viewed as a *logical storage node*. Let us first consider RAID-6, which is double-fault tolerant. Here, we consider the RAID-6 implementation based on Reed-Solomon codes [26], as shown in Figure 2(a). We divide the file into two native chunks (i.e., A and B) of size $M/2$ each. We add two code chunks formed by the linear combinations of the native chunks. Suppose now that Node 1 is down. Then the proxy must download the same number of chunks as the original file from two other nodes (e.g., B and $A + B$ from Nodes 2 and 3, respectively). It then reconstructs and stores the lost chunk A on the new node. The total storage size is $2M$, while the repair traffic is M.

We now consider the double-fault tolerant implementation of F-MSR in a proxy-based setting, as shown in Figure 2(b). F-MSR divides the file into four native chunks, and constructs eight distinct code chunks $P_1, \cdots, P_8$ formed by different linear combinations of the native chunks. Each code chunk has the same size $M/4$ as a native chunk. Any two nodes can be used to recover the original four native chunks. Suppose Node 1 is down. The proxy collects one code chunk from each surviving node, so it downloads three code chunks of size $M/4$ each. Then the proxy regenerates two code chunks P'_1 and P'_2 formed by different linear combinations of the three code chunks. Note that P'_1 and P'_2 are still linear combinations of the native chunks. The proxy then writes P'_1 and P'_2 to the new node. In F-MSR, the storage size is $2M$ (as in RAID-6), but the repair traffic is $0.75M$, which is 25% of saving.

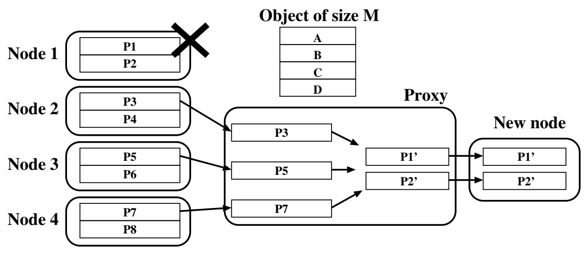
To generalize F-MSR for n storage nodes, we divide a file of size M into $2(n-2)$ native chunks, and generate $4(n-2)$ code chunks. Then each node will store two code chunks of size $\frac{M}{2(n-2)}$ each. Thus, the total storage size is $\frac{Mn}{n-2}$. To repair a failed node, we download one chunk from each of $n-1$ nodes, so the repair traffic is $\frac{M(n-1)}{2(n-2)}$. In contrast, for RAID-6, the total storage size is also $\frac{Mn}{n-2}$, while the repair traffic is M. When n is large, F-MSR can save the repair traffic by close to 50%.

Note that F-MSR keeps only code chunks rather than native chunks. To access a single chunk of a file, we need to download and decode the entire file for the particular chunk. Nevertheless, F-MSR is acceptable for long-term archival applications, whose read frequency is typically low [6]. Also, to restore backups, it is natural to retrieve the entire file rather than a particular chunk.

This paper considers the baseline RAID-6 implementation using Reed-Solomon codes. Its repair method is to reconstruct the whole file, and is applicable for *all* erasure codes in general. Recent studies [18, 28, 29] show that data reads can be minimized specifically for XOR-based erasure codes. For example, in RAID-6, data reads can be reduced by 25% compared to reconstructing the whole file [28, 29]. Although such approaches are sub-optimal (recall that F-MSR can save up to 50% of repair traffic in RAID-6), their use of efficient XOR operations can be of practical interest.



(a) RAID-6



(b) F-MSR

Figure 2: Examples of repair operations in RAID-6 and F-MSR with n = 4 and k = 2.

3 F-MSR Implementation

In this section, we present a systematic approach for implementing F-MSR. We specify three operations for F-MSR on a particular file object: (1) file upload; (2) file download; (3) repair. A key difference of our implementation from prior theoretical studies is that we do not require storage nodes to have encoding capabilities, so our implementation can be compatible with today's cloud storage. Another key design issue is that instead of simply generating random linear combinations for code chunks (as assumed in [8]), we also guarantee that the generated linear combinations always satisfy the MDS property to ensure data availability, even after iterative repairs. Here, we implement F-MSR as an MDS code for general (n,k). We assume that each cloud repository corresponds to a logical storage node.

3.1 File Upload

To upload a file F, we first divide it into $k(n-k)$ equal-size native chunks, denoted by $(F_i)_{i=1,2,\cdots,k(n-k)}$. We then encode these $k(n-k)$ native chunks into $n(n-k)$ code chunks, denoted by $(P_i)_{i=1,2,\cdots,n(n-k)}$. Each P_i is formed by a linear combination of the $k(n-k)$ native chunks. Specifically, we let $\mathbf{EM} = [\alpha_{i,j}]$ be an $n(n-k)\times k(n-k)$ encoding matrix for some coefficients α_{ij} (where $i = 1,\ldots,n(n-k)$ and $j = 1,\ldots,k(n-k)$) in the Galois field GF(2^8). We call a row vector of $\mathbf{EM}$ an *encoding coefficient vector (ECV)*, which contains $k(n-k)$ elements. We let ECV$_i$ denote the i^{th} row vector of $\mathbf{EM}$. We compute each P_i by the scalar product of ECV$_i$ and the native chunk vector (F_i), i.e., $P_i = \sum_{j=1}^{k(n-k)} \alpha_{ij} F_j$ for $i = 1, 2, \cdots, n(n-k)$, where all arithmetic operations are performed over GF(2^8). The code chunks are then evenly stored in the n storage nodes, each having $(n-k)$ chunks. Also, we store the whole $\mathbf{EM}$ in a metadata object that is then replicated to all storage nodes (see Section 4). There are many ways of constructing $\mathbf{EM}$, as long as it satisfies the MDS property and the repair MDS property (see Section 3.3). Note that the implementation details of the arithmetic operations in Galois Fields are extensively discussed in [15].

3.2 File Download

To download a file, we first download the corresponding metadata object that contains the ECVs. Then we select any k of the n storage nodes, and download the $k(n-k)$ code chunks from the k nodes. The ECVs of the $k(n-k)$ code chunks can form a $k(n-k)\times k(n-k)$ square matrix. If the MDS property is maintained, then by definition, the inverse of the square matrix must exist. Thus, we multiply the inverse of the square matrix with the code chunks and obtain the original $k(n-k)$ native chunks. The idea is that we treat F-MSR as a standard Reed-Solomon code, and our technique of creating an inverse matrix to decode the original data has been described in the tutorial [22].

3.3 Iterative Repairs

We now consider the repair of F-MSR for a file F for a permanent single-node failure. Given that F-MSR regenerates different chunks in each repair, one challenge is to ensure that the MDS property still holds even after *iterative repairs*. This is in contrast to regenerating the *exact* lost chunks as in RAID-6, which guarantees the invariance of the stored chunks. Here, we propose a *two-phase* checking heuristic as follows. Suppose that the $(r-1)^{th}$ repair is successful, and we now consider how to operate the r^{th} repair for a single permanent node failure (where $r \geq 1$). We first check if the new set of chunks in all storage nodes satisfies the MDS property after the r^{th} repair. In addition, we also check if another new set of chunks in all storage nodes still satisfies the MDS property after the $(r+1)^{th}$ repair, should another single permanent node failure occur (we call this the *repair MDS property*). We now describe the r^{th} repair as follows.

Step 1: Download the encoding matrix from a surviving node. Recall that the encoding matrix $\mathbf{EM}$ specifies the ECVs for constructing all code chunks via linear combinations of native chunks. We use these ECVs for our later two-phase checking heuristic. Since we embed $\mathbf{EM}$ in a metadata object that is replicated, we can simply download the metadata object from one of the surviving nodes.

Step 2: Select one random ECV from each of the $n-1$ surviving nodes. Note that each ECV in $\mathbf{EM}$ corresponds to a code chunk. We randomly pick one ECV from each of the $n-1$ surviving nodes. We call these ECVs to be ECV_{i_1}, ECV_{i_2}, $\cdots$, $\text{ECV}_{i_{n-1}}$.

Step 3: Generate a repair matrix. We construct a $(n-k) \times (n-1)$ *repair matrix* $\mathbf{RM} = [\gamma_{i,j}]$, where each element $\gamma_{i,j}$ (where $i = 1, \ldots, n-k$ and $j = 1, \ldots, n-1$) is randomly selected in GF(2^8). Note that the idea of generating a random matrix for reliable storage is consistent with that in [24].

Step 4: Compute the ECVs for the new code chunks and reproduce a new encoding matrix. We multiply $\mathbf{RM}$ with the ECVs selected in Step 2 to construct $n-k$ new ECVs, denoted by $\text{ECV}'_i = \sum_{j=1}^{n-1} \gamma_{i,j} \text{ECV}_{i_j}$ for $i = 1, 2, \cdots, n-k$. Then we reproduce a new encoding matrix, denoted by $\mathbf{EM}'$, which is given by:

$$i^{th} \text{ row vector of } \mathbf{EM}' = \begin{cases} \text{ECV}_i, & i \text{ is a surviving node,} \\ \text{ECV}'_i, & i \text{ is a new node.} \end{cases}$$

Step 5: Given $\mathbf{EM}'$*, check if both the MDS and repair MDS properties are satisfied.* Intuitively, we verify the MDS property by enumerating all $\binom{n}{k}$ subsets of k nodes to see if each of their corresponding encoding matrices forms a full rank. For the repair MDS property, we check that for *any* failed node (out of n nodes), we can collect *any* one out of $n-k$ chunks from the other $n-1$ surviving nodes and reconstruct the chunks in the new node, such that the MDS property is maintained. The number of checks performed for the repair MDS property is at most $n(n-k)^{n-1}\binom{n}{k}$. If n is small, then the enumeration complexities for both MDS and repair MDS properties are manageable. If either one phase fails, then we return to Step 2 and repeat. We emphasize that Steps 1 to 5 only deal with the ECVs, so their overhead does not depend on the chunk size.

Step 6: Download the actual chunk data and regenerate new chunk data. If the two-phase checking in Step 5 succeeds, then we proceed to download the $n-1$ chunks that correspond to the selected ECVs in Step 2 from the $n-1$ surviving storage nodes to NCCloud. Also, using the new ECVs computed in Step 4, we regenerate new chunks and upload them from NCCloud to a new node.

Remark: We claim that in addition to checking the MDS property, checking the repair MDS property is essential for iterative repairs. We conduct simulations to justify that checking the repair MDS property can make iterative repairs sustainable. In our simulations, we consider multiple rounds of permanent node failures for different values of n. Specifically, in each round, we randomly pick a node to permanently fail and trigger a repair. We say a repair is *bad* if the loop of Steps 2 to 5 is repeated over 10 times. We observe that without checking the repair MDS property, we see a bad repair very quickly, say after no more than 7 and 2 rounds for n = 8 and n = 12, respectively. On the other hand, checking the repair MDS property makes iterative repairs sustainable for hundreds of rounds for different values of n, and we do not yet find any bad repair after extensive simulations.

4 NCCloud Design and Implementation

We implement NCCloud as a proxy that bridges user applications and multiple clouds. Its design is built on three layers. The *file system layer* presents NCCloud as a mounted drive, which can thus be easily interfaced with general user applications. The *coding layer* deals with the encoding and decoding functions. The *storage layer* deals with read/write requests with different clouds.

Each file is associated with a *metadata* object, which is replicated at each repository. The metadata object holds the file details and the coding information (e.g., encoding coefficients for F-MSR).

NCCloud is mainly implemented in Python, while the storage schemes are implemented in C for better efficiency. The file system layer is built on FUSE [12]. The coding layer implements both RAID-6 and F-MSR. RAID-6 is built on zfec [30], and our F-MSR implementation mimics the optimizations made in zfec for a fair comparison.

Recall that F-MSR generates multiple chunks to be stored on the same repository. To save the request cost overhead (see Table 1), multiple chunks destined for the same repository are aggregated before upload. Thus, F-MSR keeps only one (aggregated) chunk per file object on each cloud, as in RAID-6. To retrieve a specific chunk, we calculate its offset within the combined chunk and issue a range GET request.

5 Evaluation

We now use our NCCloud prototype to evaluate RAID-6 (based on Reed-Solomon codes) and F-MSR in multiple-cloud storage. In particular, we focus on the setting $n = 4$ and $k = 2$. We expect that using $n = 4$ clouds may suffice for practical deployment. Based on this setting, we allow data retrieval with at most two cloud failures.

The goal of our experiments is to explore the practicality of using F-MSR in multiple-cloud storage. Our evaluation consists of two parts. We first compare the monetary costs of using RAID-6 and F-MSR based on the price plans of today's cloud vendors. We also empirically evaluate the response time performance of our NCCloud prototype atop a local cloud and also a commercial cloud vendor.

5.1 Cost Analysis

Table 1 shows the monthly price plans for three major vendors as of September 2011. For Amazon S3, we take the cost from the first chargeable usage tier (i.e., storage

	S3	RS	Azure
Storage (per GB)	\$0.14	\$0.15	\$0.15
Data transfer in (per GB)	free	free	free
Data transfer out (per GB)	\$0.12	\$0.18	\$0.15
PUT,POST (per 10K requests)	\$0.10	free	\$0.01
GET (per 10K requests)	\$0.01	free	\$0.01

Table 1: Monthly price plans (in US dollars) for Amazon S3 (US Standard), Rackspace Cloud Files and Windows Azure Storage, as of September, 2011.

usage within 1TB/month; data transferred out more than 1GB/month but less than 10TB/month).

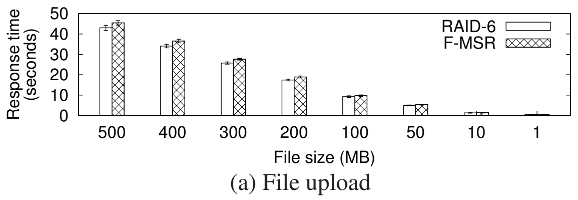
From the analysis in Section 2, we can save 25% of the download traffic during storage repair when n = 4. The storage size and the number of chunks being generated per file object are the same in both RAID-6 and F-MSR (assuming that we aggregate chunks in F-MSR as described in Section 4). However, in the analysis, we have ignored two practical considerations: the size of metadata (Section 4) and the number of requests issued during repair. We now argue that they are negligible and that the simplified calculations based only on file size suffice for real-life applications.

Metadata size: Our implementation currently keeps the F-MSR metadata size within 160B, regardless of the file size. NCCloud aims at long-term backups (see Section 2), and can be integrated with other backup applications. Existing backup applications (e.g., [27, 11]) typically aggregate small files into a larger data chunk in order to save the processing overhead. For example, the default setting for Cumulus [27] creates chunks of around 4MB each. The metadata size is thus usually negligible.

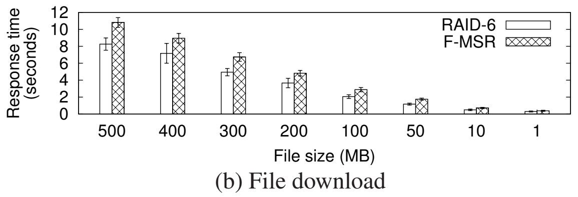
Number of requests: From Table 1, we see that some cloud vendors nowadays charge for requests. RAID-6 and F-MSR differ in the number of requests when retrieving data during repair. Suppose that we store a file of size 4MB with $n = 4$ and $k = 2$. During repair, RAID-6 and F-MSR retrieve two and three chunks, respectively (see Figure 2). The cost overhead due to the `GET` request for RAID-6 is at most 0.427%, and that for F-MSR is at most 0.854%, a mere 0.427% increase.

5.2 Response Time Analysis

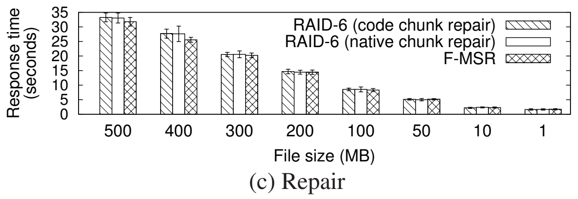
We deploy our NCCloud prototype in real environments. We then evaluate the response time performance of different operations in two scenarios. The first part analyzes in detail the time taken by different NCCloud operations, and is done on a local cloud storage in order to lessen the effects of network fluctuations. The second part evaluates how NCCloud actually performs in commercial clouds. All results are averaged over 40 runs.



(a) File upload



(b) File download



(c) Repair

Figure 3: Response times of main NCCloud operations.

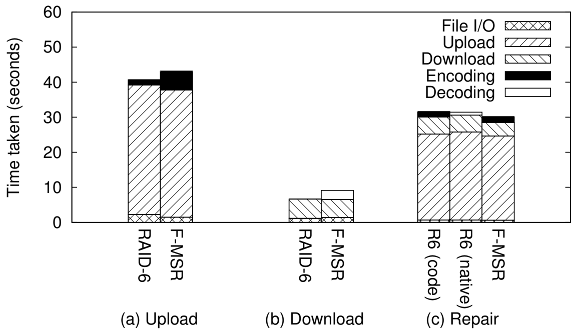
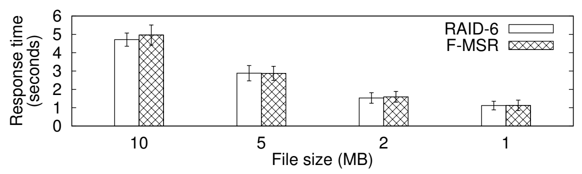


Figure 4: Breakdown of response time when dealing with 500MB file.

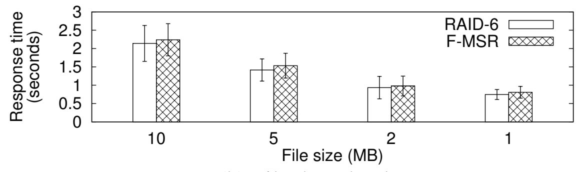
5.2.1 On a Local Cloud

The experiment on a local cloud is carried out on an object-based storage platform based on OpenStack Swift 1.4.2 [21]. NCCloud is mounted on a machine with Intel Xeon E5620 and 16GB RAM. This machine is connected to an OpenStack Swift platform attached with a number of storage servers, each with Intel Core i5-2400 and 8GB RAM. We create $(n+1) = 5$ containers on Swift, so each container resembles a cloud repository (one of them is a spare node used in repair).

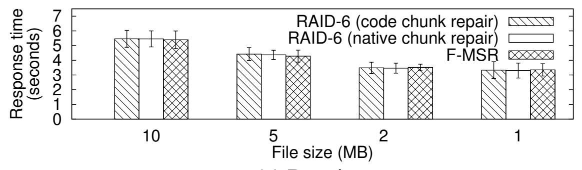
In this experiment, we test the response time of three basic operations of NCCloud: (a) file upload; (b) file download; (c) repair. We use eight randomly generated files from 1MB to 500MB as the data set. We set the path of a chosen repository to a non-existing location to simulate a node failure in repair. Note that there are two types of repair for RAID-6, depending on whether the failed node contains a native chunk or a code chunk.



(a) File upload



(b) File download



(c) Repair

Figure 5: Response times of NCCloud on Azure.

Figure 3 shows the response times of all three operations (with 95% confidence intervals plotted), and Figure 4 shows five key constituents of the response time when dealing with a 500MB file. Figure 3 shows that RAID-6 has less response time in file upload and download. With the help of Figure 4, we pinpoint the overhead of F-MSR over RAID-6. Due to having the same MDS property, RAID-6 and F-MSR exhibit similar data transfer time during upload/download. However, F-MSR displays a noticeable encoding/decoding overhead over RAID-6. When uploading a 500MB file, RAID-6 takes 1.490s to encode while F-MSR takes 5.365s; when downloading a 500MB file, no decoding is needed in the case of RAID-6 as the native chunks are available, but F-MSR takes 2.594s to decode.

On the other hand, F-MSR has slightly less response time in repair. The main advantage of F-MSR is that it needs to download less data during repair. In repairing a 500MB file, F-MSR spends 3.887s in download, while the native-chunk repair of RAID-6 spends 4.832s.

Although RAID-6 generally has less response time than F-MSR in a local cloud environment, we expect that the encoding/decoding overhead of F-MSR can be easily masked by network fluctuations over the Internet, as will be shown next.

5.2.2 On a Commercial Cloud

The following experiment is carried out on a machine with Intel Xeon E5530 and 16GB RAM running 64-bit Ubuntu 9.10. We repeat the three operations in Section 5.2.1 on four randomly generated files from 1MB to 10MB atop Windows Azure Storage. On Azure, we create $(n+1) = 5$ containers to mimic different cloud repositories. The same operation for both RAID-6 and F-MSR are run interleaved to lessen the effect of network fluctuation on the comparison due to different times of the day. Figure 5 shows the results for different file sizes with 95% confidence intervals plotted. Note that although we have used only Azure in this experiment, actual usage of NCCloud should stripe data over different vendors and locations for better availability guarantees.

From Figure 5, we do not see distinct response time differences between RAID-6 and F-MSR in all operations. Furthermore, on the same machine, F-MSR takes around 0.150s to encode and 0.064s to decode a 10MB file (not shown in the figures). These constitute roughly 3% of the total upload and download times (4.962s and 2.240s respectively). Given that the 95% confidence intervals for the upload and download times are 0.550s and 0.438s respectively, network fluctuation plays a bigger role in determining the response time. Overall, we demonstrate that F-MSR does not have significant performance overhead over our baseline RAID-6 implementation.

6 Related Work

There are several systems proposed for multiple-cloud storage. HAIL [5] provides integrity and availability guarantees for stored data. DEPSKY [4] addresses Byzantine Fault Tolerance by combining encryption and erasure coding for stored data. RACS [1] uses erasure coding to mitigate vendor lock-ins when switching cloud vendors. It retrieves data from the cloud that is about to be failed and moves the data to the new cloud. Unlike RACS, NCCloud excludes the failed cloud in repair. All the above systems are based on erasure codes, while NCCloud considers regenerating codes with an emphasis on storage repair.

Regenerating codes (see survey [9]) exploit the optimal trade-off between storage cost and repair traffic. Existing studies mainly focus on theoretical analysis. Several studies (e.g., [10, 13, 19]) empirically evaluate random linear codes for peer-to-peer storage. However, their evaluations are mainly based on simulations. NCFS [17] implements regenerating codes, but does not consider MSR codes that are based on linear combinations. Here, we consider the F-MSR implementation, and perform empirical experiments in multiple-cloud storage.

7 Conclusions

We present NCCloud, a multiple-cloud storage file system that practically addresses the reliability of today's cloud storage. NCCloud not only achieves fault tolerance of storage, but also allows cost-effective repair when a cloud permanently fails. NCCloud implements a practical version of the functional minimum storage regenerating code (F-MSR), which regenerates new chunks during repair subject to the required degree of data redundancy. Our NCCloud prototype shows the effectiveness of F-MSR in accessing data, in terms of monetary costs and response times. The source code of NCCloud is available at **http://ansrlab.cse.cuhk.edu.hk/software/nccloud**.

8 Acknowledgments

We would like to thank our shepherd, James Plank, and the anonymous reviewers for their valuable comments. This work was supported by grant AoE/E-02/08 from the University Grants Committee of Hong Kong.

References

[1] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. RACS: A Case for Cloud Storage Diversity. In *Proc. of ACM SoCC*, 2010.

[2] R. Ahlswede, N. Cai, S.-Y. R. Li, and R. W. Yeung. Network Information Flow. *IEEE Trans. on Information Theory*, 46(4):1204–1216, Jul 2000.

[3] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A View of Cloud Computing. *Communications of the ACM*, 53(4):50–58, 2010.

[4] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. DEPSKY: Dependable and Secure Storage in a Cloud-of-Clouds. In *Proc. of ACM EuroSys*, 2011.

[5] K. D. Bowers, A. Juels, and A. Oprea. HAIL: A High-Availability and Integrity Layer for Cloud Storage. In *Proc. of ACM CCS*, 2009.

[6] B. Chen, R. Curtmola, G. Ateniese, and R. Burns. Remote Data Checking for Network Coding-Based Distributed Storage Systems. In *Proc. of ACM CCSW*, 2010.

[7] CNNMoney. Amazon's cloud is back, but still hazy. `http://money.cnn.com/2011/04/25/technology/amazon_cloud/index.htm`.

[8] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran. Network Coding for Distributed Storage Systems. *IEEE Trans. on Information Theory*, 56(9):4539–4551, Sep 2010.

[9] A. G. Dimakis, K. Ramchandran, Y. Wu, and C. Suh. A Survey on Network Codes for Distributed Storage. *Proc. of the IEEE*, 99(3):476–489, Mar 2011.

[10] A. Duminuco and E. Biersack. A Practical Study of Regenerating Codes for Peer-to-Peer Backup Systems. In *Proc. of IEEE ICDCS*, 2009.

[11] B. Escoto and K. Loafman. Duplicity. `http://duplicity.nongnu.org/`.

[12] FUSE. `http://fuse.sourceforge.net/`.

[13] C. Gkantsidis and P. Rodriguez. Network coding for large scale content distribution. In *Proc. of INFOCOM*, 2005.

[14] GmailBlog. Gmail back soon for everyone. `http://gmailblog.blogspot.com/2011/02/gmail-back-soon-for-everyone.html`.

[15] K. M. Greenan, E. L. Miller, and T. J. E. Schwarz. Optimizing Galois Field Arithmetic for Diverse Processor Architectures and Applications. In *Proc. of IEEE MASCOTS*, 2008.

[16] Y. Hu, Y. Xu, X. Wang, C. Zhan, and P. Li. Cooperative recovery of distributed storage systems from multiple losses with network coding. *IEEE JSAC*, 28(2):268–276, Feb 2010.

[17] Y. Hu, C.-M. Yu, Y.-K. Li, P. P. C. Lee, and J. C. S. Lui. NCFS: On the Practicality and Extensibility of a Network-Coding-Based Distributed File System. In *Proc. of NetCod*, 2011.

[18] O. Khan, R. Burns, J. Plank, and C. Huang. In Search of I/O-Optimal Recovery from Disk Failures. In *USENIX HotStorage*, 2011.

[19] M. Martaló, M. Picone, M. Amoretti, G. Ferrari, and R. Raheli. Randomized Network Coding in Distributed Storage Systems with Layered Overlay. In *Information Theory and Application Workshop*, 2011.

[20] E. Naone. Are We Safeguarding Social Data? `http://www.technologyreview.com/blog/editors/22924/`, Feb 2009.

[21] OpenStack Object Storage. `http://www.openstack.org/projects/storage/`.

[22] J. S. Plank. A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems. *Software - Practice & Experience*, 27(9):995–1012, Sep 1997.

[23] C. Preimesberger. Many data centers unprepared for disasters: Industry group, Mar 2011. `http://www.eweek.com/c/a/IT-Management/Many-Data-Centers-Unprepared-for-Disasters-Industry-Group-772367/`.

[24] M. O. Rabin. Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance. *Journal of the ACM*, 36(2):335–348, Apr 1989.

[25] K. V. Rashmi, N. B. Shah, P. V. Kumar, and K. Ramchandran. Explicit Construction of Optimal Exact Regenerating Codes for Distributed Storage. In *Proc. of Allerton Conference*, 2009.

[26] I. Reed and G. Solomon. Polynomial Codes over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[27] M. Vrable, S. Savage, and G. Voelker. Cumulus: Filesystem backup to the cloud. In *Proc. of USENIX FAST*, 2009.

[28] Z. Wang, A. Dimakis, and J. Bruck. Rebuilding for Array Codes in Distributed Storage Systems. In *IEEE GLOBECOM Workshops*, 2010.

[29] L. Xiang, Y. Xu, J. Lui, Q. Chang, Y. Pan, and R. Li. A Hybrid Approach to Failed Disk Recovery Using RAID-6 Codes: Algorithms and Performance Evaluation. *ACM Trans. on Storage*, 7(3):11, 2011.

[30] zfec. `http://pypi.python.org/pypi/zfec`.

Extracting Flexible, Replayable Models from Large Block Traces

V. Tarasov[1], S. Kumar[1], J. Ma[2], D. Hildebrand[3], A. Povzner[3], G. Kuenning[2], and E. Zadok[1]
[1]*Stony Brook University,* [2]*Harvey Mudd College, and* [3]*IBM Almaden Research*

Abstract

I/O traces are good sources of information about real-world workloads; replaying such traces is often used to reproduce the most realistic system behavior possible. But traces tend to be large, hard to use and share, and inflexible in representing more than the exact system conditions at the point the traces were captured. Often, however, researchers are not interested in the precise details stored in a bulky trace, but rather in some statistical properties found in the traces—properties that affect their system's behavior under load.

We designed and built a system that (1) extracts many desired properties from a large block I/O trace, (2) builds a statistical model of the trace's salient characteristics, (3) converts the model into a concise description in the language of one or more synthetic load generators, and (4) can accurately replay the models in these load generators. Our system is modular and extensible. We experimented with several traces of varying types and sizes. Our concise models are 4–6% of the original trace size, and our modeling and replay accuracy are over 90%.

1 Introduction

Traces are a time-honored way to collect information about real-world workloads. The information contained in traces allows a workload to be characterized using factors such as the exact size and offset of each I/O request, read/write ratio, ordering of requests, etc. By replaying a trace, users can evaluate real-world system behavior, optimize a system based on that behavior, and compare the performance of different systems [21, 23, 25, 34].

Despite the benefits of traces, they are hard to use in practice. A trace collected on one system cannot easily be scaled to match the characteristics of another. It is difficult to modify traces systematically, e.g., by changing one workload parameter but leaving all others constant. Traces are hard to describe and compare in terms that are easily understood by system implementors. Large trace files are time-consuming to distribute and can affect the system's behavior during replay by polluting the page cache or causing an I/O bottleneck [20].

In reviewing related work, we observed that in many cases replaying the exact trace is not required. Instead, it is often sufficient to use a synthetic workload generator that accurately reproduces certain specific properties. For example, a particular system might be more sensitive to the read-write ratio than to operation size. In this situation one does not really need to replay the trace precisely; a synthetic workload that emulates that read-write ratio would suffice. Of course, this example is simplistic, and in many cases one would be interested in more complex combinations of the workload parameters. However, the general idea that only some properties of the trace affect system behavior remains valid.

Because many systems respond only to a few parameters, researchers have developed many benchmarks and synthetic workload generators, such as IOzone [7], Filebench [12], and Iometer [33], which avoid many of the deficiencies of traces. But it can be difficult to configure a benchmark so that it produces a realistic workload; simple ones are not sufficiently flexible, while powerful ones like Filebench offer so many options that it can be daunting to select the correct settings.

In this work we propose to fill the gap between traces and benchmarks by converting traces into the languages of the benchmarks. We focus here on block traces due to their relative simplicity, but we plan to extend this work to other trace types, e.g., file system and NFS.

Our system creates a universal representation of the trace, expressed as a multi-dimensional matrix in which each dimension represents the statistical distribution of a trace parameter or a function. Each parameter is chosen to represent a specific workload property. We implemented the most commonly used properties, such as I/O size, inter-arrival time, seek distance, read-write ratio, etc. End users can easily add new ones as desired. For each benchmark, a small plugin converts the universal trace matrix into the specific benchmark's language.

Many workloads vary significantly during the tracing period. To address this issue, our system supports trace *chunking* across time. Within each chunk, the workload is considered to be stable and uniform and is expressed as a separate matrix. We use chunk deduplication to save space in periods where the workload is the same.

We evaluated the accuracy of our system by generating models from several publicly available traces. We first replayed each trace on a test system, observing throughput, latency, I/O queue length and utilization, power consumption, request sizes, CPU and memory usage, and the numbers of interrupts and context switches. Then we emulated the trace by running benchmarks with generated parameters on the same system, collected the same observations, and compared the results.

Our error was less than 10% on average, and 15% at most; it can be controlled by varying several parameters. For a basic set of metrics, we converted a 1.4GB trace to the Filebench language in only 30s. The resulting trace description was 60MB, or 23.3× smaller.

2 Background and Motivation

Statistics Matter. Trace replay is a common evaluation technique because, unlike any other testing method, by definition traces represent reality. However, this realism comes at a price: the trace represents one instance of one system at one point in time. The next day's workload will inevitably be different, as will the same workload on a system with different hardware, competing workloads, etc. In the worst case, these variations might cause a system to be unintentionally optimized for an atypical operating point. Even if a trace accurately represents a target workload, rapid changes in hardware performance make it difficult to evaluate a design on a modern machine using measurements and traces captured on a different system only a few years earlier.

Our key observation is that for many purposes, *statistics* are what matter. The exact ordering of operations, their precise timing, the blocks or files accessed, and many other details recorded in a trace are variable and would change if it were re-recorded. Thus, when we replay a trace, we do not necessarily want to reproduce every detail as precisely as possible; instead, we would like to accurately represent its statistical properties.

An advantage of thinking of traces statistically is that they become much more flexible. For example, a trace collected a decade ago would record accesses to only a fraction of the blocks on a modern disk, and at a very different rate. Compared to a bulky trace, a statistical description is much simpler to scale to a modern machine and therefore provides a convenient abstraction for performing systematic evaluation of many systems.

Generating a good description requires representative trace properties to be selected. In general, the most appropriate properties depend on the system being tested, so it is impossible to create a complete list. For most purposes, however, the parameters of interest are well defined and widely adopted, e.g., I/O rate and distribution, read/write ratio. Thus, a statistical model of a trace should be able to capture those parameters, and should be able to describe them in sufficient detail so that no important information is lost. In particular, we should not reduce complex, empirically observed distributions to overly simple mathematical models, such as Poisson arrival processes, without justification.

Some workloads may also exhibit nonstandard, or even undiscovered, properties that might alter system behavior. It is therefore advisable to preserve the original traces to ensure these properties are retained. A workload generator can be adapted to include such characteristics once they are identified.

System Response. To evaluate a system empirically, workloads are applied and appropriate metrics measure its response. Performance is often characterized by throughput, latency, CPU utilization, I/O queue length, and memory usage [39,45]. Power consumption characterizes energy efficiency [29,36].

In many papers, these metrics are summarized by statistics such as averages or distributions. But as we argue above, it is often possible to accurately evaluate these metrics without resorting to a full and detailed trace replay. If the system response to a trace emulation is similar to that of a full replay, then emulation can replace full replay without biasing the results.

To evaluate the accuracy of our trace extraction and modeling system, we surveyed papers in Usenix FAST conferences from 2008–2011 and noted that the frequently used metrics fell into four categories: (1) throughput and latency; (2) I/O utilization and average I/O queue length; (3) CPU utilization and memory usage; and (4) power consumption. Most of the surveyed papers included 1–2 of these metrics, but in our study we evaluate all four types to ensure a comprehensive comparison. We claim that if all response metrics are similar, then the trace is modeled properly. We feel that our set of metrics is sufficiently representative and comprehensive to produce reliable results. There is still a chance that an unmeasured response parameter may differ; but our system is modular and easily extensible to emulate any additional metrics one desires.

Replay Methods. We use system response to evaluate our trace emulation accuracy. However, a system's response depends on the replay method, and varies based on the goal of the study. To study peak performance, traces are often accelerated [31,40,44,48]. For power efficiency, traces are usually replayed verbatim to preserve realistic idle periods [5,9]. To stress specific subsystems, a subset of the trace is sometimes replayed [38]. Our workload models can emulate existing trace-replay methods as well as more sophisticated ones.

3 Design

Our five design goals, in decreasing priority, are:

1. **Accuracy**: Ensure that trace replay and trace emulation yield matching evaluation results.
2. **Flexibility**: First, leverage existing powerful workload generators, rather than creating new ones. Therefore, traces should be translated into models that can be accurately described using the capabilities of existing benchmarks. Second, allow users to choose anything from accurate yet bulky models to smaller but less precise ones.
3. **Extensibility**: Allow the model to include additional properties chosen by the user.
4. **Conciseness**: The resulting model should be much smaller than the original trace.
5. **Speed**: The time to translate large traces should be reasonable even on a modest machine.

Feature Extraction. The first step in our model-building process is to extract important features from the trace. We first discuss how we extract parameters from workloads whose statistical characteristics do not change over time, i.e., stationary workloads. Then we describe how to emulate a non-stationary workload.

Each block trace record has a set of fields to describe the parameters of a given request. Fields may include the operation type, offset or block number, I/O size, timestamp, etc. Our translator is field-oblivious: it considers every parameter as a number. We designate these parameters as an n-dimensional vector $\vec{p} = (p_1, p_2, ..., p_n)$. We define a *feature function* vector on $\vec{p}$:

$$\vec{f} = (f_1(\vec{p}, s_1), f_2(\vec{p}, s_2), ..., f_m(\vec{p}, s_m)) = \vec{f}(\vec{p}, s_f)$$

Each feature function represents an analysis of some property of the trace; s_i represents private state data for the i-th feature function, which lets us define features across multiple trace entries and parameters.

For example, assume that p_1 and p_2 represent the I/O size and offset fields, respectively. We can then define the simple feature functions f_1—just the I/O size itself—and f_2—the logarithmic inter-arrival distance (offset difference between two consecutive requests):

$$f_1 = f_1(\vec{p}, s_1) = p_1$$

$$f_2 = f_2(\vec{p}, s_2) = \log(p_2 - s_2.\mathit{prev\_offset})$$

In our translator, the user first chooses a set of m feature functions. Evaluating these functions on a single trace record results in a vector that represents a point in an m-dimensional feature space. The translator divides the feature space into buckets of user-specified size, and collects a histogram of feature occurrences in a multi-dimensional matrix—the *feature matrix*—that explicitly captures the relevant statistics of the workload, and implicitly records their correlations.

For example, using the two feature functions above, plus a third that encodes the operation (0 for reads, 1 for writes), the resulting feature matrix might look like the one in Figure 1. In this case, the trace held 52 requests of size less than 4KB and inter-arrival distance less than 1KB; of those, 38 were reads and 14 were writes.

By choosing a set of feature functions, users can adjust the workload representation to capture any important trace features. By selecting an appropriate bucket granularity, users can control the accuracy of the representation, trading off precision for computational complexity in the translator and matrix size. Stage 1 in Figure 2 shows the translator's role in the overall design.

Once the feature matrix has been created, the translator can perform a number of additional operations on it: projection, summation along dimensions, computation of conditional probabilities, and normalization. These operations can be used by the benchmark plugins (described below) to calculate parameters. For example, using the matrix in Figure 1, a plugin might first sum across the distance-vs.-size plane to calculate the total numbers of reads and writes, normalize these to find P(read), and then generate benchmark code to conditionalize I/O size on the operation type.

Clearly, the choice of feature functions affects the quality of the emulation; currently the investigator must do this based on the insight into the particular system of interest, e.g., whether it has been optimized for certain workloads that can be reflected in an appropriate feature function. We have implemented a library of over a dozen standard feature functions based on those commonly found in the literature [10, 11, 26, 30], including operation type, I/O size, offset distribution, inter-arrival distance, inter-arrival time, process identifier, etc. New feature functions can easily be added as needed to capture specialized system characteristics.

Benchmark Plugins. Once a feature matrix has been constructed from a trace, it is possible to use it directly as input to a workload generator. However, our goal in this research is not to create yet another generator. Instead, we believe that it is best to build on the work of others by using existing workload generators and benchmarks. This approach allows us to easily reuse all the extensive facilities that these benchmarks provide. Many existing benchmarks offer a way to configure the workload that they generate; some offer command-line configuration parameters (e.g., IOzone [7] and Iometer [33]) while others offer a more extensive language for that purpose (e.g., Filebench [12] and fio [13]).

Most existing benchmarks use statistical models to generate a workload. Some of them use average parame-

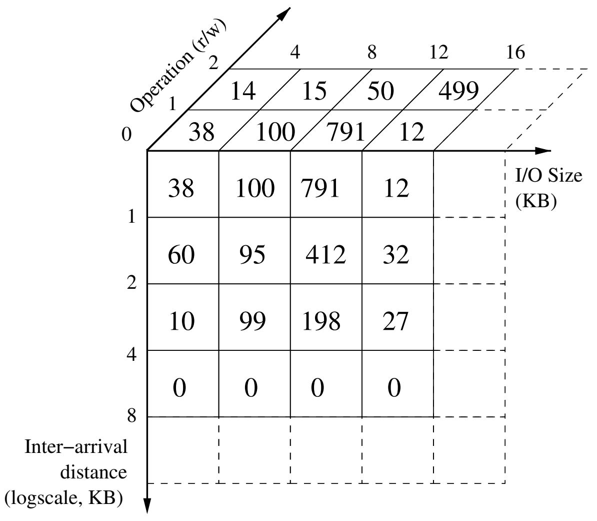


Figure 1: Workload representation using a feature matrix

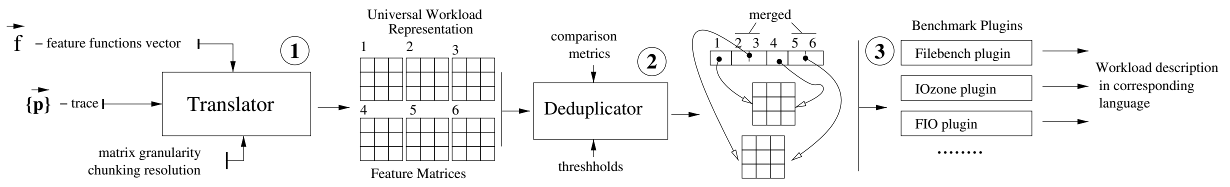


Figure 2: Overall System Design

ter values; others use more complex distributions. In all cases, our feature matrices contain all the information needed to control the models used by these benchmarks. A simple plugin translates the feature matrix into a specific benchmark's parameters or language. For some benchmarks, the expressiveness of the parameters might limit the achievable accuracy, but even then the plugin will help choose the best settings to emulate the original trace's workload. Stage 3 in Figure 2 demonstrates the role of the benchmark plugins in the overall design.

For our initial investigations, we have implemented plugins for Filebench and IOzone. We chose Filebench for its flexibility, and IOzone because it is more suitable for micro-benchmarking. We found that it was easy to add a plugin for a new benchmark, since only a single function has to be registered with the translator. The size of the function depends on the number of feature functions and the complexity of the target benchmark.

Chunking. Many real-world traces are non-stationary: their statistical characteristics vary over time. This is especially true for traces that cover several hours, days, or weeks. However, most workload generators apply a stationary load, and cannot vary it over time. We address this issue with *trace chunking*: splitting a trace into chunks by time, such that the statistics of any given chunk are relatively stable. Finding chunk boundaries is difficult, so we first use a constant user-defined chunk size, measured in seconds. For each chunk, we compute a feature matrix independently; this results in a sequence of matrices. We then convert these fixed chunks into variable-sized ones by feeding the matrices to a deduplicator that merges adjacent similar matrices (Stage 2 in Figure 2). This optimization works well because many traces remain stable for extended periods before shifting to a different workload mode. We normalize the matrices before comparing them, so that the absolute number of requests in a chunk does not affect the comparison. We use the maximum distance between matrix cells as a metric of similarity. When two matrices are found to be similar, we average their values and use the result to represent the workloads in the corresponding time chunks.

Besides detecting varying workload phases, the deduplication process also reduces the model size. To achieve even further compression, we support all-ways deduplication: every chunk in a trace is deduplicated against every other chunk (not just adjacent ones).

Along with the matrices, we generate a time-to-matrices map that serves as an additional input to the benchmark plugins. If the target benchmark is unable to support a multi-phase workload, the plugin generates multiple invocations with appropriate parameters.

In the example in Figure 2, we set the trace duration to 60s and the initial chunk size to 10s, so the translator generated six matrices. After all-ways deduplication, only two remained.

4 Implementation

Traces from different sources often have different formats. We wanted our translator to be efficient and portable. We chose the efficient and flexible DataSeries format [2]—recommended by the Storage Networking Industry Association (SNIA)—and we selected SNIA's draft block-trace semantics [37]. We wrote converters to allow experimentation with existing traces in other formats. We also created a block-trace replayer for DataSeries, which supports several commonly used replay modes. In total we wrote about 3,700 LoC: 1,500 in the translator, 800 in the converters, 1,000 in the DataSeries replayer, and 400 in the Filebench and IOzone plugins. We plan to release these publicly.

5 Evaluation

To evaluate the accuracy, conversion speed, and compression of our system, we used multiple micro-benchmarks and a variety of real traces. In this paper we present evaluation results based on two traces: Finance1 [28] and MS-WBS [22]. The Finance1 trace captures the activity of several OLTP applications running at two large financial institutions. The MS-WBS traces were collected from daily builds of the Microsoft Windows Server operating system. The high-level characteristics of the traces are presented in Table 1.

It is fair to assume that the accuracy of our translator might depend on the system under evaluation. In our experiments we used a spectrum of block devices:

Characteristic	**Finance1**	**MS-WBS**
Duration	12 hours	1.5 hours
Reads/Writes (10^6)	1.2/4.1	0.7/0.6
Avg I/O size	3.5KB	20KB
Seq. Requests	11 %	47%

Table 1: High-level characteristics of the used traces

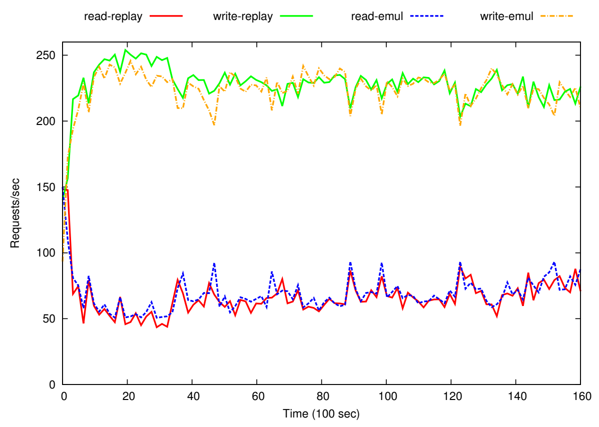


Figure 3: Reads and writes per second, Setup P, Fin1 trace.

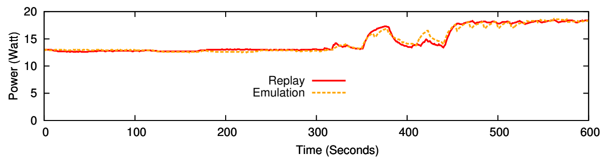


Figure 4: Disk power consumption, Setup P, MS-WBS trace.

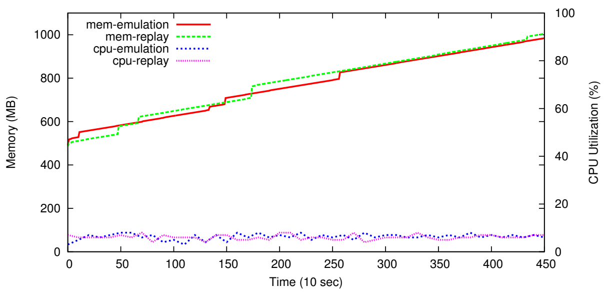


Figure 5: Memory and CPU usage, Setup P, Fin1 trace.

various disk drives, flash drives, RAIDs, and even virtual block devices. In this paper we present results from two extremes of the spectrum. In the first experimental setup—*Setup P*—we used a *P*hysical machine with an external SCSI Seagate Cheetah 300GB disk drive connected through an Adaptec 39320 controller. The fact that the drive was powered externally allowed us to measure its power consumption using a WattsUp meter [43].

The second experimental setup (*Setup V*) is an enterprise-class system that has a *V*irtual machine running under the VMware ESX 4.1 Hypervisor. The VM accesses its virtual disks on an NFS server backed by a GPFS parallel file system [19, 35]. The VM runs CentOS 6.0; the ESX and GPFS servers are IBM System x3650's, with GPFS using a DS4700 storage controller. Accuracy metrics were recorded at the NFS/GPFS server.

On both setups, we first replayed traces and then emulated them using Filebench. In all experiments we set the chunk size to 20s and enabled all feature functions. We chose the matrix granularity for each dimension experimentally, by gradually decreasing it until the accuracy began to drop. During all runs we collected the accuracy parameters specified in Section 2 using the *iostat*, *vmstat*, and *wattsup* tools; we plotted graphs showing the value of each accuracy parameter versus time for both replay and emulation. Due to limited space, we only present the graphs for a few representative accuracy parameters. However, we give the average and maximum emulation error for all experiments.

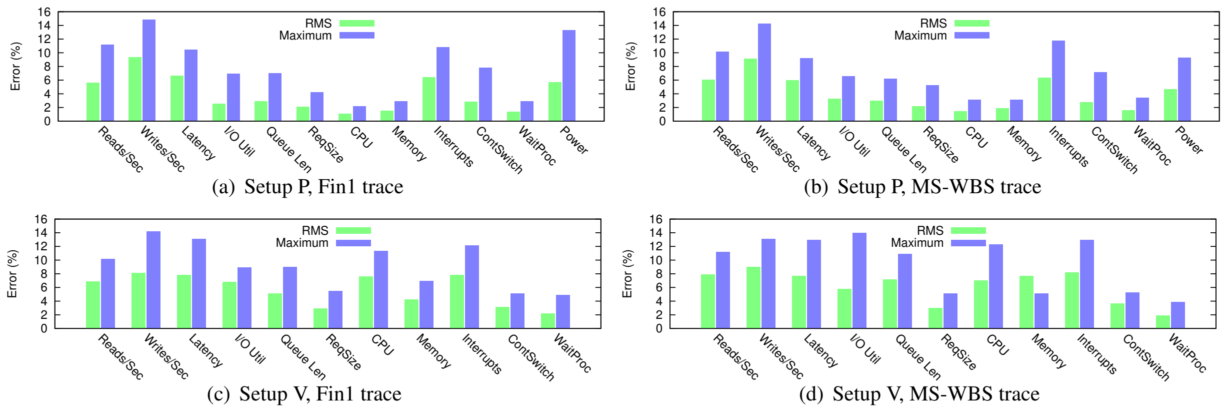
Figure 3 depicts how the throughput—for both reads and writes—changes with time for the Finance1 trace. The replay was performed with infinite acceleration; it took about 5 hours to complete on Setup P. The trace emulation line closely follows the replay line; the Root Mean Square (RMS) distance is lower than 6% and the maximum distance is below 15%. In the beginning of the run, read throughput was 4 times higher then later in the trace. By inspecting the model we found that the workload exhibits high sequentiality in the beginning of the trace. After startup, the read throughput falls to 50–100 ops/s, which is reasonable for an OLTP-like workload and our hardware. The write performance is 2–2.5 times higher than for read, due to the controller's write-back cache that makes writes more sequential.

Figure 4 depicts disk-drive power consumption in Setup P during a 10-minute non-accelerated replay and emulation of the MS-WBS trace. In the first 5 minutes trace activity was low, resulting in low power usage. Later, a burst of random disk requests increased power consumption by almost 40%. The emulation line deviates from the replay line by an average of 6%.

In Setup V, the GPFS server was caching requests coming from a virtual machine. As a result, the run time of the Fin1 trace was only 75 minutes. The memory and CPU consumption of the GPFS server during this time are shown in Figure 5. Memory usage rises steadily, increasing by about 500MB by the end of the run, which is the working-set size of the Fin1 trace. Discrepancies between replay and emulation are within 10%, but there are visible deviations at times when the memory usage steps up. We attribute this to the complexity of the GPFS's cache policy, which is affected by a workload parameter that we did not emulate. CPU utilization remained steadily about 10% for both replay and emulation.

Figure 6 summarizes the errors for all parameters, for both setups and traces. The maximum emulation error was below 15% and RMS distance was 10% on average. Although the maximum discrepancy might seem high, Figure 3 shows sufficient behavioral accuracy.

The selection of feature matrix dimensions is vital for achieving high accuracy. If a system is sensitive to a workload property that is missing in the feature matrix, accuracy can suffer. For example, disk- and SSD-based storage systems may have radically different queuing and prefetching policies. To ensure high-fidelity replays



(a) Setup P, Fin1 trace (b) Setup P, MS-WBS trace (c) Setup V, Fin1 trace (d) Setup V, MS-WBS trace

Figure 6: Root Mean Square (RMS) and maximum relative distances of accuracy parameters for two traces and two systems.

across both types of systems, the feature matrix should capture the impact of appropriate parameters.

The chunk size and matrix granularity also affect the model's accuracy. Our general strategy is to select these parameters liberally at first (e.g., 100s chunk size and 1MB granularity for I/O size) and then gradually and repeatedly restrict them (e.g., 10s chunk size, 1KB I/O size) as needed until the desired accuracy is achieved. One can always be guaranteed to get high enough accuracy if sufficiently small numbers are used.

Conversion Speed and Model Size. The speed of conversion and the size of the resulting model depend on the trace length and the translator parameters. On our 2.5GHz server, traces were converted at about 50MB/s, which is close to the throughput of the 7200RPM disk drive. The resulting model without deduplication was of approximately 10–15% size of the original trace. Deduplication removed over 60% of the chunks in both the Fin1 and MS-WBS traces, resulting in a final model size reduction of 94–96%. All sizes were measured after compressing both traces and models using gzip.

6 Related Work

The body of research related to traces is large; we cite only a representative sample. Many studies have focused on accurate trace collection with minimum interference [1, 4, 24, 31, 32]. Other researchers have proposed trace-replaying frameworks at different layers in the storage stack [3,20,48,48,49]. Since a trace contains information about the workload applied to the system, a number of works focused on trace-driven workload characterization [22, 23, 25, 34]. N. Yadwadkar proposed to identify an application based on its trace [46].

After a workload is characterized, a few researchers have suggested a workload model that allows them to generate synthetic workloads with identical characteristics [6, 14–18, 41, 42, 47]. These works address only one or two workload properties, whereas we present a general framework for any number of properties. Also, we chunk data and generate workload expressions for the languages of already existing benchmarks.

The two projects most closely related to ours are Distiller [27] and Chen's Workload Analyzer [8]. Distiller's main goal is to identify important workload properties. We can use this information to intelligently define dimensions for our feature matrix. Chen uses machine learning techniques to identify the dependencies between workload features. However, the authors do not emulate traces based on the extracted information.

7 Conclusions and Future Work

We have created a system that extracts flexible workload models from large I/O traces. Through the novel use of chunking, we support traces with time-varying statistical properties. In addition, trace extraction is tunable, allowing model accuracy and size to be traded off against creation time. Existing I/O benchmarks can readily use the generated model by implementing a plugin. Our evaluation with Filebench and several block traces demonstrated that the accuracy of generated models approaches 95%, while the model size is less than 6% of the original trace size. Such concise models allow easy comparison, scaling and other modifications.

In the future we plan to support file-system-level traces, build multi-layer models, and add flexibility in the analysis phase. Our current chunking method is simple and we want to investigate alternative chunking techniques. We will also work on a graphical tool for manual trace chunking. To avoid manual selection of the translator's parameters, we want to explore various artificial intelligence approaches. To further reduce the model size, we plan to improve the compression ratio by matching empirical distributions in the feature matrix to explicit mathematical functions. We recognize that our list of accuracy metrics is not complete and want to experiment with other accuracy parameters (e.g., latency distributions). We also plan to develop tools and techniques that will simplify various operations on our models, such as time and size scaling, and comparison to other models.

References

[1] E. Anderson. Capture, conversion, and analysis of an intense NFS workload. In *Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FAST '09)*, 2009.

[2] E. Anderson, M. Arlitt, C. Morrey, and A. Veitch. DataSeries: an efficient, flexible, data format for structured serial data. *ACM SIGOPS Operating Systems Review*, 43(1), January 2009.

[3] E. Anderson, M. Kallahalla, M. Uysal, and R. Swaminathan. Buttress: A toolkit for flexible and high fidelity I/O benchmarking. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST '04)*, 2004.

[4] A. Aranya, C. P. Wright, and E. Zadok. Tracefs: a file system to trace them all. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST '04)*, 2004.

[5] T. Bisson, S.A. Brandt, and D.D.E. Long. A hybrid disk-aware spin-down algorithm with I/O subsystem support. In *Proceedings of the IEEE 2007 Performance, Computing, and Communications Conference (IPCCC)*, 2007.

[6] P. Bodik, A. Fox, M. Franklin, M. Jordan, and D. Patterson. Characterizing, modeling, and generating workload spikes for stateful services. In *Proceedings of the First ACM Symposium on Cloud Computing (SOCC)*, 2010.

[7] D. Capps. IOzone file system benchmark. `www.iozone.org`.

[8] Y. Chen, K. Srinivasan, G. Goodson, and R. Katz. Design implications for enterprise storage systems via multi-dimensional trace analysis. In *Proceedings of the 23rd ACM Symposium on Operating System Principles (SOSP '11)*, 2011.

[9] F. Douglis, P. Krishnan, and B. Bershad. Adaptive disk spin-down policies for mobile computers. In *Proceedings of the Second Symposium on Mobile and Location-Independent Computing*, 1995.

[10] M. Ebling and M. Satyanarayanan. SynRGen: An extensible file reference generator. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1994.

[11] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS tracing of email and research workloads. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST '03)*, 2003.

[12] Filebench. `http://filebench.sourceforge.net`.

[13] fio—flexible I/O tester. `http://freshmeat.net/projects/fio/`.

[14] A. Ganapathi, Y. Chen, A. Fox, R. Katz, and D. Patterson. Statistics-driven workload modeling for the cloud. In *Proceedings of the International Workshop on Information and Software as Services (WISS)*, 2010.

[15] G. Ganger. Generating representative synthetic workloads: an unsolved problem. In *Proceedings of Computer Measurement Group Conference (CMG)*, 1995.

[16] M. Gomez and V. Santonja. A new approach in the modeling and generation of synthetic workloads. In *Proceedings of the 8th Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2000.

[17] B. Hong and T. Madhyastha. The relevance of long-range dependence in disk traffic and implications for trace synthesis. In *Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST)*, 2005.

[18] B. Hong, T. Madhyastha, and B. Zhang. Cluster-based input/output trace analysis. In *Proceedings of 24th IEEE International Performance, Computing, and Communications Conference (IPCCC)*, 2005.

[19] IBM. IBM scale out metwork attached storage. `www.ibm.com/systems/storage/network/sonas/`.

[20] N. Joukov, T. Wong, and E. Zadok. Accurate and efficient replaying of file system traces. In *Proceedings of the Fourth USENIX Conference on File and Storage Technologies (FAST '05)*, 2005.

[21] S. Kavalanekar, D. Narayanan, S. Sankar, E. Thereska, K. Vaid, and B. Worthington. Measuring database performance in online services: a trace-based approach. In *Proceedings of TPC Technology Conference on Performance Evaluation and Benchmarking (TPC TC)*, 2009.

[22] S. Kavalanekar, B. Worthington, Q. Zhang, and V. Sharda. Characterization of storage workload traces from production windows servers. In *Proceedings of IEEE International Symposium on Workload Characterization (IISWC)*, 2008.

[23] T. Kimbrel, A. Tomkins, R. Patterson, B. Bershad, P. Cao, E. Felten, G. Gibson, A. Karlin, and K. Li. A trace-driven comparison of algorithms for parallel prefetching and caching. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI 1996)*, 1996.

[24] A. Konwinski, J. Bent, J. Nunez, and M. Quist. Towards an I/O tracing framework taxonomy. In *In Proceedings of the International Workshop on Petascale Data Storage (PDSW)*, 2007.

[25] G. H. Kuenning, G. J. Popek, and P. Reiher. An analysis of trace data for predictive file caching in mobile computing. In *Proceedings of the Summer 1994 USENIX Conference*, 1994.

[26] Z. Kurmas. *Generating and Analyzing Synthetic Workloads using Iterative Distillation*. PhD thesis, Georgia Institute of Technology, 2004.

[27] Z. Kurmas, K. Keeton, and K. Mackenzie. Synthesizing representative I/O workloads using iterative distillation. In *Proceedings of IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems (MASCOTS)*, 2003.

[28] LASS. UMass trace pepository. `http://traces.cs.umass.edu`.

[29] T. Li and L. K. John. Run-time modeling and estimation of operating system power consumption. In *Proceedings of the 2003 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 2003.

[30] Z. Li, Z. Chen, S. M. Srinivasan, and Y. Zhou. C-Miner: Mining block correlations in storage systems. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST '04)*, 2004.

[31] M. P. Mesnier, M. Wachs, R. R. Sambasivan, e. Lopez, J. Hendricks, G. R. Ganger, and D. O'Hallaron. //TRACE: parallel trace replay with approximate causal events. In *Proceedings of the Fifth USENIX Conference on File and Storage Technologies (FAST '07)*, 2007.

[32] R. Moore. A universal dynamic trace for Linux and other operating systems. In *Proceedings of the 2001 USENIX Annual Technical Conference (ATC)*, 2001.

[33] OSDL. Iometer project. `www.iometer.org`.

[34] J. Ousterhout, H. Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of the Tenth ACM Symposium on Operating System Principles (SOSP)*, 1985.

[35] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST '02)*, 2002.

[36] P. Sehgal, V. Tarasov, and E. Zadok. Evaluating performance and energy in file system server workloads extensions. In *Proceedings of the Eighth USENIX Conference on File and Storage Technologies (FAST '10)*, 2010.

[37] Storage Networking Industry Association (SNIA). Block I/O trace common semantics (working draft). `www.snia.org/sites/default/files/BlockIOSemantics-v1.0r11.pdf`, February 2010.

[38] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger. Metadata efficiency in versioning file systems. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST '03)*, 2003.

[39] A. Traeger, N. Joukov, C. P. Wright, and E. Zadok. A Nine Year Study of File System and Storage Benchmarking. *ACM Transactions on Storage (TOS)*, 4(2):25–80, May 2008.

[40] B. Trushkowsky, P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. The SCADS director: scaling a distributed storage system under stringent performance requirements. In *Proceedings of the Nineth USENIX Conference on File and Storage Technologies (FAST '11)*, 2011.

[41] M. Wang, A. Ailamaki, and C. Faloutsos. Capturing the spatio-temporal behavior of real traffic data. In *Proceedings of Performance*, 2002.

[42] M. Wang, T. Madhyastha, N. Chan, and S. Papadimitriou. Data mining meets performance evaluation: fast algorithms for modeling burst traffic. In *Proceedings of 16th International Conference on Data Engineering (ICDE)*, 2002.

[43] Watts up? PRO ES Power Meter. `www.wattsupmeters.com/secure/products.php`.

[44] C. Weddle, M. Oldham, J. Qian, A. A. Wang, P. Reiher, and G. Kuenning. PARAID: a gear-shifting power-aware RAID. In *Proceedings of the Fifth USENIX Conference on File and Storage Technologies (FAST '07)*, 2007.

[45] S. Wu, H. Jiang, D. Feng, L. Tian, and B. Mao. WorkOut: I/O workload outsourcing for boosting RAID reconstruction performance. In *Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FAST '09)*, 2009.

[46] N. Yadwadkar, C. Bhattacharyya, and K. Gopinath. Discovery of application workloads from network file traces. In *Proceedings of the Eighth USENIX Conference on File and Storage Technologies (FAST '10)*, 2010.

[47] J. Zhang, A. Sivasubramaniam, H. Franke, N. Gautam, Y. Zhang, and S. Nagar. Synthesizing representative I/O workloads for TPC-H. In *Proceedings of International Sypmposium on High Performance Computer Architecture (HPCA)*, 2004.

[48] N. Zhu, J. Chen, and T. Chiueh. TBBT: scalable and accurate trace replay for file server evaluation. In *Proceedings of the Fourth USENIX Conference on File and Storage Technologies (FAST '05)*, 2005.

[49] N. Zhu, J. Chen, T. Chiueh, and D. Ellard. An NFS trace player for file system evaluation. Technical Report TR-14-03, Harvard University, December 2003.

scc: Cluster Storage Provisioning Informed by Application Characteristics and SLAs

Harsha V. Madhyastha[*], *John C. McCullough*[†], *George Porter*[†], *Rishi Kapoor*[†], *Stefan Savage*[†], *Alex C. Snoeren*[†], *and Amin Vahdat*[†]
UC Riverside[*] and UC San Diego[†]

Abstract

Storage for cluster applications is typically provisioned based on rough, qualitative characterizations of applications. Moreover, configurations are often selected based on rules of thumb and are usually homogeneous across a deployment; to handle increased load, the application is simply scaled out across additional machines and storage of the same type. As deployments grow larger and storage options (e.g., disks, SSDs, DRAM) diversify, however, current practices are becoming increasingly inefficient in trading off cost versus performance.

To enable more cost-effective deployment of cluster applications, we develop *scc*—a storage configuration compiler for cluster applications. *scc* automates cluster configuration decisions based on formal specifications of application behavior and hardware properties. We study a range of storage configurations and identify specifications that succinctly capture the trade-offs offered by different types of hardware, as well as the varying demands of application components. We apply *scc* to three representative applications and find that *scc* is expressive enough to meet application Service Level Agreements (SLAs) while delivering 2–4.5$\times$ savings in cost on average compared to simple scale-out options. *scc*'s advantage stems mainly from its ability to configure heterogeneous—rather than conventional, homogeneous—cluster architectures to optimize cost.

1 Introduction

Today, application providers can choose from a range of storage choices to provision the infrastructure for cluster-based applications. Storage technologies as diverse as DRAM, solid state drives (SSDs), and hard disks present complex trade-offs in cost, capacity, performance (along multiple dimensions), and power consumption. New storage technologies such as phase change memory [14] will soon further complicate the space.

Provisioning, however, is based largely on rules of thumb and best practices. Applications are broadly categorized as storage, compute, or memory intensive and are typically deployed on homogeneous clusters heavy on the corresponding resource. As application load increases, deployments are "scaled out" by simply adding more storage and compute in the same configuration. Not only does this state of affairs fail to take full advantage of the diversity of available storage choices, but the increasing scale of deployments makes such inefficiencies worse; inefficiencies multiplied over thousands of servers can have substantial costs. In the scale-out model, a poor initial choice can greatly inflate expenses.

In this paper, we pursue an alternate approach—the automated selection of cluster storage configurations based on formal specifications of applications, hardware, and workloads. Initially, such an approach places significant burden on those developing and deploying applications to characterize applications and workloads. However, the resultant savings in cost necessary to satisfy Service Level Agreements (SLAs) can be substantial.

Our primary contributions in implementing this approach are two-fold. First, we determine how the characteristics of applications, workloads, and hardware should be specified in order to automate the selection of cluster configurations. To do so, we study several representative deployment scenarios and identify a parsimonious yet sufficiently expressive set of parameters that capture the trade-offs offered by different types of storage devices and the varying demands across application components. Though others have pursued a similar approach of formally specifying workloads and hardware [5, 7, 34], we extend this approach to account for various types of storage media (e.g., disk, SSD, and DRAM) and to jointly capture storage and compute requirements of applications. We show that it is feasible to concisely summarize the most salient parameters that determine the resource requirements of specific application deployments, thus minimizing the burden of formal specification.

Second, we develop *scc*, a storage configuration compiler that takes specifications of applications, workloads,

Resource	MB/s	IOPS	Watts	Cost
7.2K Disk (500 GB)	90 (R) 90 (W)	125 (R) 125 (W)	5	$213
15K Disk (146 GB)	150 (R) 150 (W)	285 (R) 285 (W)	2.3	$296
SSD (32 GB)	250 (R) 80 (W)	2500 (R) 1000 (W)	2.4	$456
DRAM (1 GB)	12.8K (R) 12.8K (W)	1.6B (R) 1.6B (W)	3.5	$35
CPU core	-	-	20	$137

Server type	Resource Limits	Cost
Server1	4 cores, 1 Gbps network 12GB DRAM, 4 SAS slots	$1400
Server2	16 cores, 10 Gbps network 48GB DRAM, 16 SAS slots	$1850
Server3	32 cores, 10 Gbps network 512GB DRAM, 16 SAS slots	$11000

Table 1: Example set of cluster building blocks input to *scc*. Cost is price plus energy costs for 3 years. *scc* takes read and write *gap* parameters as input rather than IOPS.

and hardware as input, automates the navigation of the large space of storage configurations, and zeroes in on the configuration that meets application SLAs at minimum cost. To evaluate *scc*, we experiment with three distributed applications with distinctly different workload characteristics: 1) ProductSearch, a product search webservice modeled on Google Merchant Center [17], 2) Terasort, a MapReduce job to sort large tuple collections, and 3) PhotoShare, a photo-sharing Web service modeled on Flickr. By deploying these applications on a range of cluster configurations and measuring application performance on these configurations, we present empirical evidence that *scc* is expressive enough to capture the needs of a range of applications.

In developing *scc* and applying it to diverse application workloads, we make three key observations. First, the right choice of storage configuration depends not only on the storage capacity and I/O needs of the application, but also on the application's compute requirements and on the types of server configurations available. When an application performs a set of operations in sequence, the resources assigned to serve each of these operations must be jointly optimized to satisfy the performance bound on the sequence of operations at minimum cost. For example, in an application that performs an I/O operation on some data followed by some computation, the storage type assigned to the data depends on the amount of computation. When the computation consumes significant time, the data may need to be stored on fast storage like SSDs to meet performance bounds, whereas when compute time is low, there is greater slack in performing the I/O and hence, slower cheaper storage like disk may suffice.

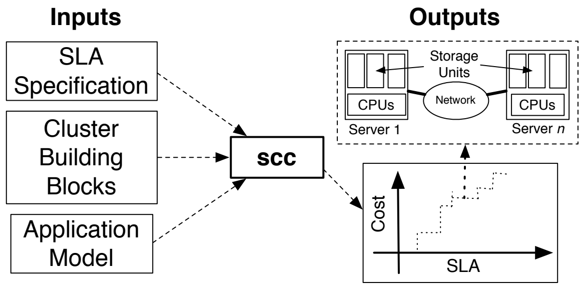


Figure 1: Overview of *scc*.

Second, we find that clusters with heterogeneity—rather than conventional homogeneity—across servers are necessary to optimize cost. The resources required differ across application components because of varying ratios of capacity, compute, and I/O throughput needs across components. For example, in a deployment of the photo-sharing Web service, it may be cheaper to store photos on disk and cache thumbnails in DRAM; storing both on disk or both in DRAM may result in higher cost due to higher I/O throughput needs from thumbnails or higher storage capacity needs of photos, respectively. As a result, *scc*'s suggested configuration meets performance SLAs at low cost. For example, in experiments with Terasort, we find that *scc* meets performance requirements at 15–20% lower cost than a homogeneous configuration recommended based on best practices.

Finally, we also find that the most cost-effective cluster architecture depends not only on the application being provisioned but also on the workload and performance requirements. Data that was initially capacity-bound may become I/O-bound at higher loads, calling for shifts from high capacity but slow storage, e.g., disks, to low capacity but fast storage, e.g., SSDs. As a result, cluster configurations output by *scc* for ProductSearch and PhotoShare result in 2x–4.5x average savings in cost compared to similarly performant scale-out options.

2 Problem setting and overview

Identifying an appropriate cluster architecture to host a large-scale service is often not straightforward. For example, given a set of resources to choose from (e.g., as shown in Table 1), an application provider has to answer several questions. What storage technologies should be employed, and how should data be partitioned across them? Where should caching be employed? What types of servers should be chosen to house the selected storage units? In addition, even if the application's implementation is efficient and there is coarse-grained parallelism in the underlying workload, how will algorithmic shifts in the application or variations in workload affect the appropriate cluster architecture? Our goal is to automate the process of answering these questions, rather than relying solely on human judgment.

Problem setting. In developing *scc*, our focus is on the typical scenario where a cluster is dedicated to a specific application, rather than large-scale data centers (e.g., Google, Microsoft) that host a mix of applications. *scc* caters to the common case where an application provider either acquires hardware or uses third-party infrastructure to deploy an application. In such cases, the question we seek to answer is: what information from the infrastructure provider and from the application developer is necessary to determine a cost-effective cluster configuration that meets performance goals?

Overview of *scc*. As shown in Figure 1, *scc* takes three inputs: i) a model of application behavior, specified by the application's developer, ii) characteristics of available hardware building blocks specified by the infrastructure provider, and iii) application performance metrics, i.e., a parameterized service level agreement (SLA). Given these inputs, *scc* computes how cluster cost varies as a function of SLA and outputs a low-cost cluster configuration that meets the SLA at each point in the space. For example, a webservice SLA might specify a peak query rate per second. For each potential SLA value (e.g., 1000 queries per second), *scc* determines a cost-effective cluster architecture capable of satisfying the SLA. *scc*'s output cost vs. SLA value distribution helps administrators decide what performance can be supported cost effectively.

Our focus in developing *scc* is to show how to systematically exploit storage diversity; i.e, select among different physical media, local and remote storage, and various caching strategies. In the future, we plan to extend *scc* to tailor network configurations and choose among CPU types. Here, we assume the cluster network can deliver uniform bandwidth between all pairs of servers [4] and do not address incast-like scenarios [27] that arise due to limited packet buffers. Instead, we assume network storage access is limited only by network adapter speeds.

```
<sla task="photoview" rate="300"> </sla>
<sla task="photoupload" rate="100"> </sla>
<sla task="tagview" rate="100"> </sla>
```

(a)

```
<resources>
  <storage_unit name="7.2KDisk" capacity="500GB" bus="SAS"
      rateR="90MBps" gapR="8ms" rateW="90MBps" gapW="8ms"
      volatile="0" price="200" power="5W"> </storage_unit>
  <storage_unit name="SSD" capacity="32GB" bus="SAS"
      rateR="250MBps" gapR="0.4ms" rateW="80MBps" gapW="1ms"
      volatile="0" price="450" power="2.4W"> </storage_unit>
  <storage_unit name="DRAM" capacity="1GB" bus="DDR3-1333"
      rateR="12.8GBps" gapR="0.6ns" rateW="12.8GBps" gapW="0.6ns"
      volatile="1" price="25" power="3.5W"> </storage_unit>
  ... additional storage units ...
  <cpu price="85" power="20W"> </cpu>
  <server name="HP DL380 G6" price="1400" cpus="4" BW="1Gbps">
      <bus name="SAS" slots="4" BW="6Gbps"> </bus>
      <bus name="DDR3-1333" slots="12" BW="21.3GBps"> </bus>
  </server>
  ... additional servers ...
</resources>
```

(b)

```
<application>
  <dataset name="tables_repository" size="150GB" persistent="1"
      remote="1"> </dataset>
  <dataset name="hot_ratingsdata" size="1.6GB" persistent="*"
      remote="0"> </dataset>
  <dataset name="cold_ratingsdata" size="6.4GB" persistent="*"
      remote="0"> </dataset>
  ... additional datasets ...
  <task name="worker" phase="exec" memory="1GB">
    <io op="R" dataset="tables_repository" record_size="800MB"
        num_records="1" blocking="0"> </io>
    ... additional I/O operations ...
    <compute time="2.2s" blocking="1"> </compute>
    <dependency task="queryprocessor" num_invocations="1"
        parallel="1" blocking="1"> </dependency>
  </task>
  <task name="queryprocessor" phase="exec" memory="200MB">
    <io op="R" dataset="hot_ratingsdata" probability="0.8"
        num_records="40K" record_size="4KB" blocking="0"> </io>
    <io op="R" dataset="cold_ratingsdata" probability="0.2"
        num_records="40K" record_size="4KB" blocking="0"> </io>
    <compute time="0.65s" blocking="1"> </compute>
  </task>
</application>
```

(c)

Figure 2: Example specifications of (a) SLAs for PhotoShare, (b) hardware resources, and (c) application behavior for a particular deployment of ProductSearch.

3 Inputs to *scc*

We now describe how we represent the three inputs to *scc*—SLA specifications, properties of cluster building blocks, and application models. Rather than model the intricate complexities of algorithms and hardware, *scc* captures aggregate high level statistics that are relevant to application and hardware scaling behavior over a broad range of scenarios. Towards this end, we identify a key set of elements that comprise each of *scc*'s inputs and the corresponding attributes required to describe these elements. Figure 2 depicts examples of *scc*'s three inputs; our implementation encodes them in XML.

3.1 Specifying SLAs

We consider throughput-based SLAs for two distinct application classes: batch and interactive; we defer supporting latency-based SLAs to future work. For batch applications, the SLA has two attributes—the job size and the required execution time, e.g., for a MapReduce job, the SLA specifies the number of records to be processed and the total run time for doing so. *scc* is more applicable for provisioning a new set of VMs for every job than for provisioning a shared cluster used for running jobs with varying I/O and compute characteristics. For interactive applications run as services, each type of request is associated with its own performance-based SLA that describes its required sustained processing rate. For example, in the case of a photo sharing Web service, the rates of photo uploads, photo views, and album views are each specified as a separate SLA. *scc*'s SLAs specify peak rather than average case throughput. We discuss how *scc* accounts for temporal variation in Section 6.3.

3.2 Cluster building blocks

scc's second input is a characterization of the set of building blocks available for assembling the cluster. We account for three types of elements—storage units, CPU cores, and servers. To ensure our approach is not tied to the characteristics of any particular technology, we employ abstract features such as I/O bandwidth and number of processor slots as the attributes for these elements. Table 1 lists sample building blocks used in our evaluation.

3.2.1 Storage

Storage resources come in discrete units, e.g., 1 disk or 1 stick of DRAM. To differentiate between different kinds of storage technologies such as disk, SSDs and DRAM, we characterize each unit based on two properties: capacity and I/O throughput. Capacity is simply the amount of available storage measured in bytes. Representing I/O throughput is more complex; we capture it with four attributes—the average *rate* at which I/O requests are served and the average latency *gap* between serving successive I/O requests, accounting for both separately for reads and writes. The gap parameter captures overheads involved with non-sequential I/O, e.g., seeks on disks and block erasure on SSDs. We define read (write) gap for a particular storage device as the latency incurred on average between successive reads (writes) to random logical addresses on the device. The latency to serve a read (write) request for a chunk of *size* bytes is thus $(\frac{size}{rate} + gap)$. We consider gap rather than the commonly used IOPS metric because gap enables us to better capture the range of I/O performance regions from small to large records. For example, characterizing read performance on a 7.2K-RPM disk based on IOPS and rate works well for 4 KB and 10 MB reads, but fails to capture the read throughput with 200 KB reads. In our evaluation, we find that these four attributes—rate and gap for reads and writes—suffice to capture the I/O performance of multiple disk types and SSDs. Furthermore, we believe these attributes are expressive enough to capture the characteristics of phase change memory (PCM) and other emerging storage technologies.

The application-visible performance of a storage medium is also influenced by how the chosen file system places data. For example, a disk can deliver significantly higher write throughput when written to in a log format [28]. Therefore, when an application stores a dataset on a storage or file system, we measure I/O rates and gaps of each storage unit when using that system to read/write data. Further, for each storage unit, we consider two other attributes: storage persistence (i.e., whether it provides non-volatile storage) and I/O bus type (e.g., SAS vs. PCIe).

3.2.2 Servers and compute

Servers impose constraints on how storage can be packed into a physical box. For each kind of server, we consider its memory capacity as well as the properties of its I/O controllers. For each I/O controller, we consider the total number of units it can support and its maximum available I/O bandwidth. For example, a serial attached SCSI (SAS) controller permits up to 128 connected disks, yet supports a maximum I/O bandwidth of only 6 Gbps, less than the total sequential I/O throughput that can be obtained from 128 disks. Similarly, throughput for remote storage is limited by a server's network interface speed.

As our focus is on storage complexity in cluster architectures, we consider only a single CPU type, ignoring trade-offs in compute per unit power [6, 11]. Instead, we vary the number of cores per server to extract the level of parallelism needed to maximize storage utilization.

3.2.3 Costs

Finally, an additional attribute for every element in the resource specification is the amortized cost per hardware unit including both capital and operational outlays. In our current implementation, the latter subsumes energy costs, ignoring data center costs and administrator salaries, and we consider total cluster cost to be a linear sum of individual components, which may not necessarily be true for large quantities. We leave for future work discounting the growth of expenses with cluster size and accounting for increased operational costs with a higher diversity of server configurations in the cluster.

3.3 Characterizing applications

Our characterization of applications accounts for two aspects—its implementation and the workload in its planned deployment. However, unlike previous attempts at formally specifying workloads [34], simply accounting for storage capacity needs and the application's stream of I/O operations does not suffice for our purpose. Instead, to capture an application's implementation, we first ask the application's developer to describe its decomposition into compute and storage components, and the interaction between them. For example, Figure 3 depicts the components, and the interaction between them, for one of the three applications we consider later in our evaluation—a photo sharing Web service, PhotoShare. Though our approach places the onus on application developers to go through the process of formally specifying the components of their application, an application's specification is reusable across deployments. Some of the characteristics of several applications are already captured today [23, 24].

Second, we enable those who deploy an application to annotate the specification of the application's architecture with properties of the expected workload in their

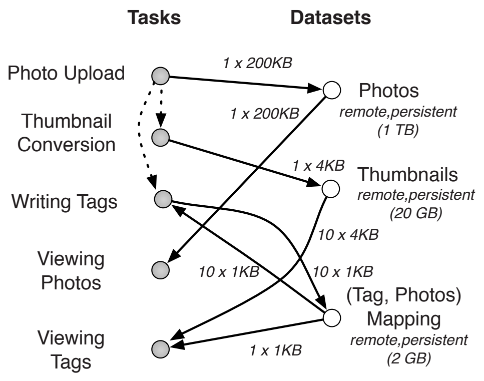


Figure 3: Interaction between tasks and datasets in example application PhotoShare. Edges between tasks and datasets represent I/O with direction differentiating input and output. Dotted edges indicate task dependencies.

deployment. To do so, we require that the compute and I/O characteristics of an application's components, when subjected to the target workload, be determined by running small-scale application benchmarks. Extracting these properties requires tracing the application's execution—now standard practice in resource-intensive performance-critical applications. In the absence of built-in tracing support, systems like Magpie [8] can be leveraged.

3.3.1 Tasks and datasets

scc's application specification separates the application's compute and storage requirements into *tasks* and *datasets*. A task is a specific application functional unit; all threads/processes that perform the same function together constitute a single task. A dataset is a collection of records of the same type with similar I/O access patterns.

Execution of tasks. To account for how compute time and I/O wait time are distributed across a task's execution, we represent each task by its *execution path*; different tasks in an application will have different execution paths. A task in an interactive application executes its execution path for each incoming request, whereas in batch applications, a task's execution path is executed as many times as necessary to consume its input. Further, since batch jobs can go through multiple phases of execution, we require the application developer to tag each task with the phase to which it belongs. The cluster can thus be provisioned to support the maximal resource requirement across phases.

We characterize the execution path of a task as a sequence of three types of operations—compute, I/O, and invocations of other tasks. Each of these can be marked as either blocking or non-blocking. Compute operations are characterized by the amount of time spent performing computation on a particular type of CPU. While this value can of course vary, we have found that a representative average is sufficient to inform *scc*; we show later in Section 6.1 that *scc* can help evaluate the sensitivity of its output to the input values. I/O operations are attributed with the dataset on which the operation is being performed and whether it is a read or write operation. Similarly, every task dependency is annotated with the invoked task.

The operations in a task's execution path may not be completely deterministic. For example, an I/O operation may hit the cache in some cases but not all, or a remote task may need to be invoked only based on the results of prior task invocations. To capture such non-determinism, every operation has an additional attribute—the probability of its execution. This, for example, enables us to capture developer knowledge of typical working set sizes for individual datasets and the hit rate on the working set.

Lastly, we also require that each task node be tagged with its memory requirements. While some applications may use all available memory and garbage collect on demand, we consider required memory to be the amount necessary to maintain performance. Note that this specifies memory that *scc* must allocate for computation beyond any additional DRAM *scc* provisions as RAM disks to store datasets.

Representing datasets. Next, we account for datasets in terms of their I/O bandwidth and capacity requirements. The I/O requirements from a dataset are determined by all the I/O operations performed on it, across the execution paths of all tasks. We ask that each I/O operation be tagged with three attributes—the number of records read or written, the number of bytes in each record, and whether records are read in parallel. The last of these three properties can be specified by the application developer, while the other two depend on the workload for which the application is being deployed. Again, we find that average values suffice for our target throughput-based SLAs. Describing I/O in terms of records accounts for the overhead seen between successive read/write operations on storage media such as disks and SSDs, e.g., from disk seeks. We similarly annotate task dependencies with three attributes—the number of invocations being performed, whether they are in parallel, and whether the whole dependency is blocking or non-blocking.

Lastly, we account for a dataset's capacity requirements by requiring that it be tagged with three additional attributes: its size, whether it must be persistent, and whether the dataset is local or remote. This last attribute differentiates between data assumed in the application's implementation to be on a storage unit local to the task accessing it as opposed to data that may be stored on a storage unit on a different machine in the cluster. Though a remote file can be made to appear local by use of systems such as NFS, we capture the application developer's assumption of local storage, since remote access leads to higher access latencies. *scc* leverages this distinction in two ways. For a remote dataset, *scc* explicitly accounts for network load resulting from I/O requests and some CPU requirements for the machines hosting the dataset. Conversely, task-local storage constrains the amount of parallelism available on a single machine due to the storage bandwidth and number of storage unit slots available on the node.

Figure 2(c) presents an example (for another of the applications we use in our evaluation, ProductSearch, a product search Web service) of the precise format in which such an application characterization is specified as input to *scc*.

4 Implementation of *scc*

Next, we describe how *scc* processes its inputs to generate cost-effective cluster configurations.

4.1 Overview

scc determines the cost versus SLA distribution for a given application deployment by considering the configuration for each point in the distribution independently. To compute the cluster configuration for a target SLA, *scc* needs to answer two questions. First, it needs to determine the *architecture* of the cluster—for each dataset of the application, it must determine the type of media on which the dataset should be stored and how to pack the storage units into servers. This packing is constrained by the number and location of CPUs available to assign to the compute tasks that access each dataset. Second, *scc* needs to identify the *scale* at which this architecture must be instantiated to meet the SLA—scale is determined by the number of servers, storage units, and CPUs, as well as the level of parallelism of each application task.

Guiding Principles. Two key principles help *scc* identify the right cluster configuration. First, the architecture and scale for every application component can be determined independently when all operations are performed asynchronously, but not when some operations are synchronous. The SLA for any task only specifies the rate at which a task's execution path must run. In the typical case where a task's execution path contains some operations that block others, *scc* needs to determine the "division of labor" across these operations that minimizes cost. For example, in a task that reads from an input dataset and then writes to an output dataset, in order to meet the task's SLA, it may suffice to provision fast storage for any one of the two datasets; provisioning fast storage for both datasets may unnecessarily result in higher cost due to storage capacity requirements, whereas slow storage for both may incur higher costs in satisfying I/O throughput needs. Hence, *scc* jointly determines resource requirements across all application components.

```
Configuration state: $S = (S_1, \ldots, S_n)$, where
  $S_i$ = storage type assigned to $i^{th}$ dataset
for every remote dataset $d_i$
  compute $U_i$ = no. of units of $S_i$ to meet capacity and
  I/O needs from $d_i$
for every task $t_i$
  $R_i$ = average runtime of $t_i$
  $P_i$ (parallelism of task $t_i$) = SLA($t_i$) $\times$ $R_i$
  for every dataset $d_j$ local to $t_i$,
    compute no. of units of $S_j$ to meet capacity and
    I/O needs from $d_j$ for one instance of $t_i$
Linear integer program to choose servers
  Variables:
    1. booleans for whether $k^{th}$ server is of $j^{th}$ type
    2. $\forall$ remote dataset $d_i$, no. of units of $S_i$ in $k^{th}$ server
    3. $\forall$ task $t_i$, no. of instances on $k^{th}$ server
  Constraints:
    Per-server constraints:
    1. On each I/O controller, (no. of storage units $<$ no. of
       slots) and (I/O throughput $<$ bus bandwidth)
    2. (I/O throughput on remote datasets and local datasets
       accessed remotely) $<$ network bandwidth
    3. no. of CPUs $<$ no. of CPU slots
    Per-dataset and per-task constraints:
    1. $\forall$ dataset $d_i$, (no. of units across all servers = $U_i$)
    2. $\forall$ task $t_i$, (no. of instances across all servers = $P_i$)
  Objective:
    Minimize cost of (servers + storage units + CPUs)
```

Figure 4: Summary of *scc*'s procedure for determining a cost-effective cluster configuration that satisfies target SLAs, given a particular assignment of storage types to datasets.

Second, since *scc* is provisioning for peak load, it prevents over-provisioning by ensuring that at least one resource is bottlenecked on every server at peak load. (If the application provider desires to run the cluster at lower peak utilization, that can be specified as input.) Based on our characterization of hardware, there are four possible bottlenecks on each server—1) the number of slots or 2) the bandwidth on an I/O controller, 3) the number of CPU cores, or 4) network bandwidth.

Algorithm. Driven by the need for joint optimization across components, *scc* represents each point in the state space of configurations by the assignment of storage unit types to datasets. As a result, if S is the number of storage choices and D is the number of datasets, *scc* has to search through a space of $O(S^D)$ configurations; for each dataset, *scc* can choose any one of the S storage options.

In cases where the configuration space is too large to perform an exhaustive search, *scc* performs a repeated gradient descent search: We start with a randomly chosen configuration. In each step, we consider all neighboring configurations—those which differ in exactly one dataset's storage-type assignment—and move to the configuration that still meets the SLA with the maximum decrease in cost. We repeat this step until we find a configuration where all neighbors have higher cost. Since gradient descent can lead to a local minimum, we repeat this procedure multiple times with different randomly chosen initial configurations and settle on the minimum cost output across the multiple attempts. In our evaluation, we have found that repeating the gradient descent 10 times is typically sufficient to find a solution close to the global minimum. Therefore, even when determining the configuration to satisfy workloads of tens of thousands of queries per second, *scc*'s running time for any particular SLA is within a minute.

At the heart of *scc*'s search of the configuration space is a procedure—summarized in Figure 4—that, given any particular assignment of storage types to datasets, determines a cost-effective set of resources to meet the target SLAs. In this procedure, *scc* first determines for each remote dataset, i.e., not local to any task, the number of storage units required of the type assigned to the dataset in the configuration state. Second, *scc* determines the number of CPUs required by every task and the number of storage units of the assigned type needed by the task's local datasets. Finally, it determines the types of servers and number of each kind required to minimize overall cluster cost. We describe these three steps using examples from illustrative applications.

4.2 Resources for datasets

A dataset's storage resources need to satisfy two requirements: capacity and I/O throughput. To determine the cheapest storage solution that satisfies both, *scc* computes the number of storage units required to satisfy each requirement independently and chooses the maximum of the two. When the former (latter) is more expensive, we call the dataset capacity (I/O) bound. A capacity-bound dataset requires storage equal to the dataset's size irrespective of the medium used. Determining the storage required by a I/O-bound dataset is more involved. Though the total capacity of the storage units allocated to the dataset need only be equal to the dataset's size, we may need more units—under-utilizing the capacity on each of them—to meet throughput demands.

We compute I/O requirements as follows. As described in Section 3.3, the application characterization specifies the record size and the number of records read/written in every I/O operation. *scc* computes the overall number of I/O operations that a particular storage unit can support based on its rate and gap parameters. The SLA combined with the probability attributed to an I/O operation fully specifies the required frequency of the operation, which in turn determines the number of storage units required to deliver the performance in parallel.

For example, when serving requests to view photos in PhotoShare, one photo of size 200 KB on average is read from the photos dataset on every photo view. If the photos dataset were assigned to 15K-RPM disk (Table 1), which offers a read rate of 150 MBps and a read gap of 3.5 ms, it will be able to serve 200 KB-sized reads at the throughput of $\frac{200\text{KB}}{\frac{200\text{KB}}{150\text{MBps}}+3.5\text{ms}}$, approximately 40 MBps. Therefore, if the SLA specifies 1000 photo views per second, $\frac{200\text{KB}\times 1000/s}{40\text{MBps}} = 5$ units of 15K-RPM disks are required to satisfy the I/O throughput requirement.

4.2.1 Task phases

Not all tasks in an application execute concurrently, e.g., the Map and Reduce tasks run in different phases of a MapReduce job. Since datasets are subject to I/O operations only from tasks executing in a particular phase, *scc* computes the storage needed to meet I/O requirements in each phase independently. The storage requirements for a dataset during a particular execution phase are computed as the sum of storage needs across all the I/O operations made on the dataset by the tasks that run in that phase. *scc* computes the overall I/O-mandated storage requirement as the maximum over all phases.

4.2.2 Caching for higher I/O

When a dataset is I/O-bound, storing it across units of a single type may not always be the cheapest solution. I/O throughput of persistent datasets can be improved by introducing a second type of storage unit as a caching layer. For example, when considering a single storage type to service the entire load, the SSD is the most cost-effective option for the tags dataset in the PhotoShare application. However, a cheaper solution is to store the persistent copy of the tags on a 7.2K-RPM disk and to serve reads from a cached copy in DRAM.

scc assumes write-through caching. Persistent storage units handle all writes and maintain a persistent copy. Units of another type, with higher I/O rates, handle all reads. To ensure durability, every write is committed to both copies and by default, *scc* provisions enough storage to cache the entire dataset. However, developer knowledge of the application's working set size—encoded into the application specification as different capacity requirements for the dataset and for the cache—can also be used to determine what fraction of the dataset is to be cached. To evaluate whether such a solution is cost effective, *scc* computes the costs of both copies of the dataset separately and computes their sum.

4.3 Task Resources

scc next determines the resource requirements of each compute task in three steps. First, it determines the CPU utilization of the task. Second, it computes the degree of parallelism—i.e., the number of threads/processes of the task—required to meet the SLA. Finally, it determines the number of storage units required per instance of the task for each of the task's local datasets.

A task's CPU utilization is the fraction of its run time spent performing computation. *scc* translates a task's CPU utilization into the corresponding CPU resources required by computing the level of parallelism required to meet the SLA: if a task's execution path is to be executed with frequency F and the task's average run time is R, then $(F \cdot R)$ instances of the task are required. The value of F for a task is computed from the SLA for that task and other tasks that depend on it; R is computed by appropriately summing up the times for compute, I/O, and task invocation operations in the task's execution path, taking into account, for each operation, its probability and whether it is blocking or non-blocking.

scc calculates each task's storage requirements for its local datasets based on capacity and I/O throughput requirements. *scc* also computes the task's memory requirements and the network bandwidth needed for I/O accesses to remote storage. *scc* determines each of these three requirements—local storage, memory, and network bandwidth—per instance of the task and linearly extrapolates to a target level of parallelism.

4.4 Optimizing server costs

Finally, *scc* optimizes cluster cost by minimizing the cost of required servers. Determining the servers required to host storage and CPU resources reduces to the multi-dimensional vector bin packing problem [12]. Each server type is associated with a cost and a vector of resource limits, such as the I/O bandwidth of each I/O controller and the maximum number of CPUs that the server can accommodate. Respecting these resource limits, CPUs and storage units required by tasks and datasets must be placed across servers, while minimizing total cost. *scc* solves this bin-packing problem with a linear integer program.

5 Evaluation

Next, we demonstrate that *scc* achieves the right cost versus performance tradeoff. Unfortunately, it is difficult to select appropriate comparisons. Though there exists a large body of work on capacity planning [22], all of it revolves around the question: "Given a cluster architecture for an application, how many servers of each type in the architecture are necessary?" In contrast, *scc* minimizes cost by determining not only the right scale, but also the architecture most suited for a given application deployment. Moreover, conversations with major infrastructure providers reveal that existing approaches for provisioning cluster applications used in practice are ad-hoc—the primary motivation for our work.

5.1 Methodology

We apply *scc* to three distributed applications with disparate workload characteristics to identify the cost-versus-SLA tradeoff in each case. To keep the discussion simple, we fix capacity requirements while varying the SLA. For each application, we validate the cost-effectiveness of *scc*'s output for one particular target SLA. Though *scc* readily outputs cluster configurations on the scale of tens of thousands of servers, we focus on smaller scales for validation so that we can instantiate the configurations with hardware we have on hand. Note that even at the scale of a few servers, the combination of type and quantity for storage, compute, and servers results in a very large configuration space. For example, with 5 servers of type Server1, over 10^{14} cluster configurations are feasible using the building blocks in Table 1.

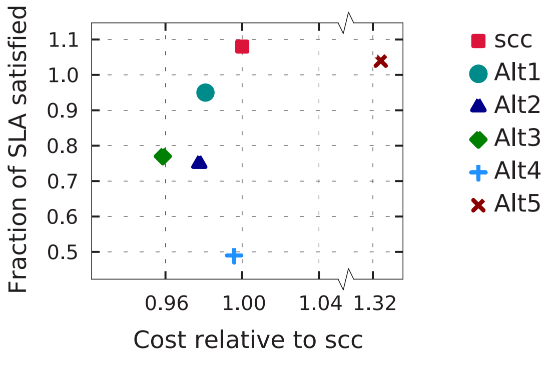
In the absence of prior approaches for principled determination of cluster architectures, our evaluation compares configurations output by *scc* with *all* possible alternative assignments of datasets to storage types; for each alternative, we consider those quantities of hardware resources to make cost comparable to *scc*. Here, we present results from alternate architectures that come closest to matching *scc* with respect to satisfaction of SLAs. In some cases, we also consider alternative architectures at the scale required to meet input SLAs and show that they incur higher costs than *scc*. For each experiment, we physically provision clusters composed of the building blocks provided as input to *scc*.

Table 1 summarizes the resources provided as input to *scc*, represented formally as in Figure 2(b). We construct our specification for cluster building blocks based on HP ProLiant DL380 G6 servers interconnected by a Gigabit Ethernet network. In each server (Server1), we consider the resource limitations to be one quad-core Intel Xeon processor, four SAS slots, and up to 12 GB of DRAM. Each of the SAS slots can support a 7.2K-RPM disk, a 15K-RPM disk, or an Intel SSD. To evaluate the performance of a given configuration, we turn off CPU cores and/or use only a subset of the SAS and DIMM slots.

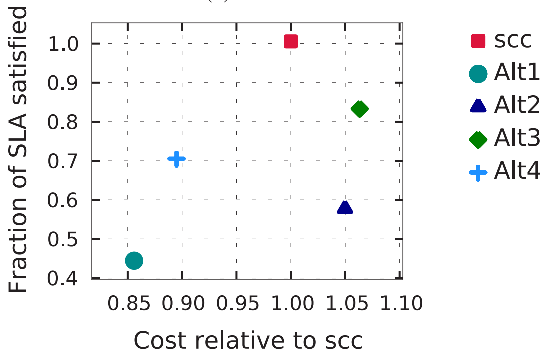
For each of the resources, we consider the cost to be the amount we paid, excluding support, plus energy costs computed based on power usage numbers from product data sheets (we assume $\$0.10/kWh$ over a three year deployment). Though the power drawn by any unit can vary from its specification, we study the robustness of our results (Section 6.1) and find that they remain unchanged even if energy costs increase by a factor of two.

5.2 Photo sharing

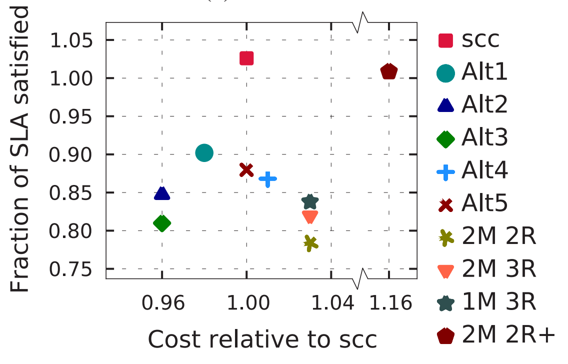
Our first application, PhotoShare, is an interactive photo sharing application. It allows users to upload tagged photos, to view thumbnails for photos associated with a given tag, and to view the photos. PhotoShare is a



(a) PhotoShare



(b) ProductSearch



(c) MapReduce Terasort

Figure 5: Validation of cluster output by *scc* for particular SLA values in the three application cases.

C++ FastCGI application hosted on lighttpd webservers. Uploaded images are thumbnailed and stored, whereas tag updates are made via RPCs. Data is kept in a distributed log-based key-value storage system. Image, tag, and thumbnail views translate to fetches from the store. The three SLA metrics are the simultaneous rates for uploading photos, viewing photos, and viewing thumbnails associated with tags. Our input workload has, on average, 200-KB images that convert to 4-KB thumbnails, and an average of 10 photos/tag and 10 tags/photo.

We apply *scc* to study the cost as a function SLA by fixing the ratio of the rates for uploads, photo views, and tag views at 1:3:1. Figure 6 shows this cost distribution for a range of SLA values. Perhaps surprisingly, no huge spikes are observed in this distribution; this is because *scc* balances costs across the kind of storage, the number of CPUs, and the number of machines provisioned. Rather than adding more machines of the same type, the cluster architecture transitions to faster storage as the SLA becomes more stringent, with transitions in storage type for different datasets seen at different SLA values. Table 2 highlights these transitions. Note that the quantity in which different types of resources are provisioned varies within each architecture regime specified by every row in the table.

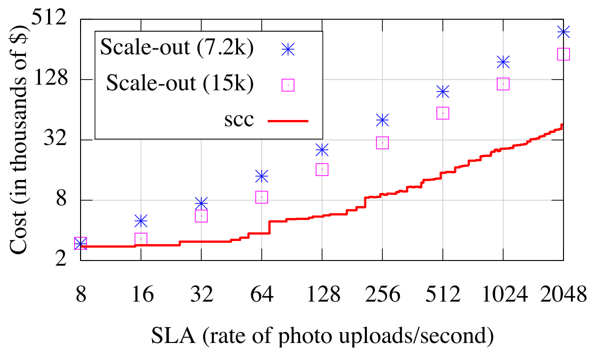


Figure 6: Cost versus SLA distribution output by *scc* for PhotoShare. Note log scale on *y* axis.

We further compare the cost output by *scc* with the cost associated with a scale-out approach. We compare the *scc* configuration to the cases where the building block is based around: 1) storage servers with four 7.2K-RPM disks (the cost-optimal storage type for all datasets at the lowest SLA), and 2) servers with four 15K-RPM disks. In either case, more storage servers are added as the required rates increase. Figure 6 shows that the costs in both cases are significantly greater than with *scc*, incurring between 3 and 4.5 times more cost (note the logarithmic *y* axis). Thus, simply scaling out a homogeneous configuration that is cost-effective at low loads can result in significant cost inflation at higher loads.

To verify the performance of *scc*'s suggested configuration, we focus on one particular SLA: 100 uploads/s, 300 photo views/s, and 100 tag views/s. The fraction of the SLA satisfied is the minimum fraction of sustained request rates across uploads, photo views, and tag views. *scc* determines the following cluster configuration for this SLA: one machine, with 4 CPU cores and 2 GB of DRAM hosts the webserver; a second machine stores the photos across four 15K-RPM disks; and a third machine hosts one SSD for thumbnails, and 1 GB of DRAM and one 7.2K-RPM disk for tags. Each of the two storage machines have 2 CPU cores and an additional 1 GB of DRAM, as required by the key-value storage system.

Figure 5(a) shows that this configuration meets

Uploads/s	Storage unit type		
	Photos	*Thumbnails*	*Tags*
≤ 5	Disk	Disk	Disk
5–25	Disk	Disk	Disk + DRAM
25–330	Disk	SSD	Disk + DRAM
330–930	SSD	Disk + DRAM	Disk + DRAM
930–10k	Disk + DRAM	Disk + DRAM	Disk + DRAM

Table 2: Different regimes based on SLA requirements in the cost-effective architecture for PhotoShare.

the SLA; in fact, the configuration is slightly over-provisioned. It also shows the configuration is near a minimum: removing a core from the webserver (*Alt1*), replacing the thumbnail's SSD with a cheaper 15K-RPM disk (*Alt2*), removing one of the photo disks (*Alt3*), or replacing the thumbnail's SSD with two 7.2K-RPM disks (*Alt4*) all result in SLA misses. A scale-out architecture extending *Alt4* with more 7.2K-RPM drives (*Alt5*) incurs 30%-higher cost to meet the SLA.

5.3 Product search

Our second application is a multi-merchant product search and comparison service, which we call ProductSearch. We store product tables, which include product serial numbers, types, descriptions, and costs, along with product type field indices in a Hadoop Distributed File System (HDFS). In addition, user rating data is stored in a separate database table. Worker processes running across the cluster process queries for the cheapest product of a given type with a minimum user-specified rating. Each worker maintains a local copy of the ratings table as well as an index on the product serial number field; the ratings table and index are hence, specified as local datasets in the application's specification. To execute a query, a worker fetches the relevant product table and index from HDFS and then performs a join with the ratings table on the product serial number field, selecting for rows with the specified product type.

In our deployment, we build product tables with an average of 200K products, each with an average of 200 ratings. This translates to 8 GB for the ratings and roughly 800 MB for each product table. The SLA for this application specifies the required query rate.

We apply *scc* to determine system cost as a function of the SLA value. As with PhotoShare, the architecture of the cost-effective cluster changes significantly across different regimes of the SLA. At low query rates, *scc* recommends disks for both HDFS and local storage of workers. As the required query rate increases, *scc* transitions to using faster storage or provisioning more machines to handle the increased load. Figure 7 illustrates one particular transition between query rate regimes. Also, in this case as well, *scc*'s configurations yield significant cost savings compared to simple scale-

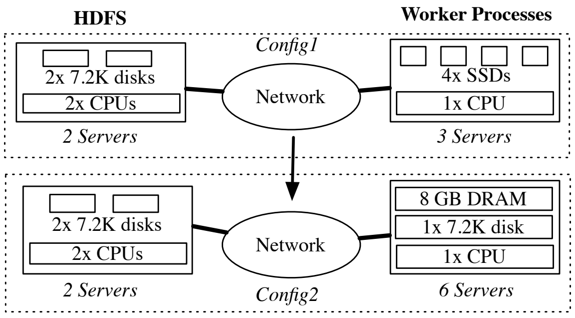


Figure 7: Transition in *scc*'s output for ProductSearch from *Config1* at 12 queries/minute to *Config2* at 13 queries/minute.

out options—roughly 3× and 2× savings on average in comparison to the scaling out of homogeneous configurations with 7.2K-RPM and 15K-RPM disks, which are cost-optimal at low loads.

We validate *scc* with an SLA of 12 queries per minute. *scc*'s cluster output for this case has two parts. First, the HDFS repository is stored across two machines, each with one CPU and two 7.2K-RPM disks. Second, 12 worker processes are spread across three machines, each with one CPU and four SSDs. We run this configuration for 15 minutes. Figure 5(b), which plots the fraction of required queries completed during the experiment, shows that this configuration is able to meet the SLA.

Next, we compare *scc*'s output with alternative configurations. First, we consider clusters with alternative local storage for the workers—*Alt1* and *Alt2* use 15K-RPM drives, and *Alt3* uses 7.2K-RPM disks with DRAM. In each case, we consider the number of workers and servers to keep cost comparable to *scc*. In both *Alt1* and *Alt2*, the disk's lower random read throughput inflates query processing times and, hence, aggregate throughput falls well below the SLA. The performance of *Alt3* comes close to the SLA, but still falls short. Second, when we place all four disks underlying HDFS into one machine (*Alt4*), the 1 Gbps network becomes a bottleneck relative to the aggregate read throughput from four 7.2K-RPM drives. As a result, download times increase, leading to SLA violations.

We also use this example application to test *scc*'s ability to capture knowledge of working set sizes. We again apply *scc* to satisfy the SLA of 12 queries per minute, but this time with the additional input that 20% of product types receive 80% of queries (the application specification for this case is shown in Figure 2(c)). In this case, *scc* outputs an alternate architecture where 12 worker processes, previously run on three machines each with four SSDs, are now instead run on three machines each with four 15K-RPM disks and 10 GB of DRAM. Queries to "hot" products are served from DRAM and those to "cold" data are served from the disks. This configuration meets the SLA with 7%-lower cost than the case where access patterns were assumed to be uniform.

5.4 Sorting binary tuples

Our final application, Terasort [29], is a MapReduce job that sorts collections of 100-byte tuples, each consisting of a 10-byte key and a 90-byte value. A *Mapper* reads tuples from a local input file and sends them over the network to appropriate *Shuffle* processes. Each Shuffler writes the tuples it receives to a set of intermediate, sorted local files. Once the Mappers and Shufflers are done, the Shuffle processes transform into the role of *Reducers*. Each Reducer merges the tuples in the local files into an output file of sorted tuples. For this application, the SLA is the total runtime of the MapReduce job.

We use *scc* to determine the cost of clusters capable of sorting 50 GB for a range of runtimes. Note that though we put together clusters of individual servers here, we envision that *scc* will be used for such jobs to provision a set of virtual machines in a virtualized infrastructure. Unlike PhotoShare and ProductSearch, we see no significant architecture changes over different runtimes. *scc* uses the basic building block of provisioning Mappers on machines with four cores and one 7.2K-RPM disk and Shufflers/Reducers on machines with four cores and two 7.2K-RPM disks. *scc* provisions more machines for both components to meet more stringent SLAs. Faster storage has no benefits because the job is CPU bound.

Next, we verify the performance of the cluster output by *scc* for an SLA that requires 50 GB to be sorted in 25 minutes—an average sorting rate of 2 GB per minute. The *scc* cluster consists of 8 Mappers and 16 Reducers spread across two and four machines respectively with the above-mentioned building blocks. We run the application on this cluster to sort 50 GB of input data. Figure 5(c) plots the SLA-specified runtime divided by the observed runtime and shows that the *scc* cluster meets the SLA.

To evaluate the cost-effectiveness of *scc*'s output, we also sort 50 GB of data on several alternative architectures. A few such alternatives include *Alt1* and *Alt2*, which reduce the number of cores from 4 to 3 on the Mapper machines and on the Reducer machines, respectively. *Alt3* substitutes the two 7.2K-RPM disks on each of the four Reducer machines with one 15K-RPM disk shared between the intermediate and output data. Figure 5(c) shows that the runtime of the Terasort job misses the SLA by at least 10% in every case. The figure also shows that two other alternatives—*Alt4* and *Alt5*—which have similar cost to *scc*'s output but trade off compute resources for more or faster storage, also fall short.

Unlike our other two example applications, compute-intensive MapReduce jobs have a cluster configuration recommended by best practices. We modify the cluster architecture to be six machines each with four cores and two 7.2K-RPM disks—a setup recommended by Cloudera for a "Balanced Compute Configuration" [13]. Also, we configure every node in the cluster to run a fixed number of Mappers and Reducers. We evaluate three different combinations of Mappers and Reducers per node (the *"2M 2R"*, *"2M 3R"*, and *"1M 3R"* points in Figure 5(c)), and interestingly, we find that the recommended MapReduce configurations deliver lower performance than *scc* for similarly priced clusters. While all three alternatives meet the SLA when scaled out to an additional machine, e.g., the *"2M 2R+"* point in the figure, this results in 16%-higher cost than *scc*'s recommendation.

Attribute	Range with same architecture		
	Lowest value	Input value	Highest value
Avg. photo size	50 KB	200 KB	850 KB
Avg. thumbnail size	1 KB	4 KB	30 KB
SSD unit price	$200	$450	$900

(a)

Dataset	Most sensitive to what change in hardware costs?
Photos	20% drop in $ of 7.2K-RPM disk
Thumbnails	92% drop in $ of DRAM
Tags	31% drop in $ of 15K-RPM disk

(b)

Table 3: Determining robustness of *scc*'s output with respect to its input: (a) robustness of cluster configuration with respect to input values for a sample set of attributes, and (b) the change in hardware costs to which *scc*'s storage decision for each dataset is most sensitive.

6 Discussion

In this section, we discuss the robustness of *scc*'s output, its utility in planning application implementation architectures, and its extensibility on other fronts.

6.1 Robustness of *scc*'s output

scc's output cluster configuration for a target SLA is a function of both the SLA and the *exact* values specified for the various attributes in the application and hardware specifications. In practice, a user of *scc* may not have precise values for all attributes due to incomplete knowledge of the application workload, uncertainty of hardware costs, or measurement inaccuracy in benchmarking.

scc is naturally built to cope with such uncertainty. For every attribute in the input specifications, *scc* varies the value of the attribute in the neighborhood of the initially specified value. For each attribute, it then outputs the range of values for that attribute wherein the cost-effective cluster architecture, i.e., the types of resources assigned to different application components, remains unchanged; variance of the attribute's value within this range can be handled by simply adding more resources of the same type. Outside of that range, the cluster will need to be revamped with a different type of resource for some application component, a significantly more cumbersome undertaking. For example, we again consider PhotoShare with an SLA of 100 uploads/s, 300 photo views/s, and 100 tag views/s. Table 3(a) shows the value ranges output by *scc* for a few attributes, within which the cluster architecture is robust to change. For example, we see that as long as average photo size remains between 50 KB and 850 KB, the cluster architecture remains the same as that obtained with the input value of 200KB.

Furthermore, *scc* can also evaluate the sensitivity of its choice of storage configuration for every dataset in the application. For example, consider PhotoShare again with the same input SLA as above. Based on current hardware costs, *scc* determines that photos be stored on 15K-RPM disks, thumbnails be stored on SSDs, and tags be stored persistently on 7.2K-RPM disks and cached in DRAM, in order to meet the SLA at minimum cost. However, these recommendations are likely to change as prices for storage units drop. *scc* can determine how robust are its choice of storage options to such changes in hardware prices. To do so, it varies the price of every type of storage unit from its input value down to 0, and notes the inflection points at which the optimal storage choice for some dataset changes. Based on this analysis, it can determine, for every dataset, that change in hardware price to which the current storage choice for the dataset is most sensitive. Table 3(b) shows the output of this analysis for the three datasets in PhotoShare. While the storage choices for photos and tags are sensitive to relatively small reductions in the prices for 7.2K-RPM and 15k-RPM disks, *scc*'s recommendation of storing thumbnails on SSDs is very robust to price fluctuations.

6.2 Informing application development

Thus far, we assumed a fixed application implementation. However, *scc* can also help determine the best application architecture. For instance, in the case of Terasort, there is a fundamental performance tradeoff between a cluster configuration with sufficient DRAM to store all data to be sorted and one that must stage portions of the data into memory from secondary storage. The former case requires one read and one write of all the data while the latter requires two reads and two writes of the data [3].

To explore cost–performance tradeoffs for the two application architectures, we must consider the benefits of servers with more network bandwidth (so remote storage does not become a bottleneck) and more memory (to allow for storing the entire dataset in memory). In Table 1, Server2 is the same HP ProLiant DL380 G6 server as Server1, but with more resources per server and a 10-Gigabit Ethernet (10GigE) NIC. Server3 is the HP ProLiant DL785 G5 Server, which accommodates more processors and DRAM, again with a 10GigE NIC.

We use *scc* to determine the cluster configuration necessary to sort 100 TB in the time required to read/write the whole data from/to disks twice at the read/write rate of the 7.2K-RPM disk. This cluster costs \$239K and completes sort in 10,000 seconds. For the alternative implementation where all data fits in DRAM, we apply *scc* to satisfy the SLA of sorting the complete dataset in half the SLA of the baseline implementation. The cheapest cluster configuration determined in this case costs \$5.6M and sorts 100 TB in 5,000 seconds. Thus, according to *scc*, the latter implementation provides a 2× speedup at 24× the cost. The application designer can decide if the faster processing is worth it.

6.3 Extensibility of *scc*

Our approach of determining cost-effective cluster configurations with *scc* is extensible in several ways.

Less flexible infrastructure services. Though we restrict our attention in this paper to flexible infrastructure services that permit arbitrary mixing and matching of compute and storage resources on a per-server or per-VM basis, *scc* can also be readily applied to less flexible services that offer only certain combinations of processor, storage, and memory configurations, e.g., Amazon's EC2 service [1]. In such cases, each combination of resources offered by the infrastructure service can be provided as input to *scc* as a separate server type, and the cost of each server will subsume the costs of all the resources that come with it.

Accounting for availability. Though we have focused on performance requirements of applications thus far, performance and availability SLAs need to be considered in unison. For example, a cheap disk type may be an attractive option for a capacity-bound dataset but the degree of replication necessary to meet availability goals may make the option cost-prohibitive. *scc* can be extended to pick for each dataset that combination of storage type and associated replication factor that meets the combination of performance, availability, and consistency requirements at minimum cost.

Load variation and incremental growth. Our current implementation of *scc* provisions applications for peak load. However, when the distribution of load across time is available, *scc* can leverage the information in two ways. First, *scc* can estimate energy costs more accurately. Second, when pricing for resources is "elastic", i.e., a user can provision resources on-demand and pay for what she uses, *scc* can make incremental reconfiguration decisions, determining when to simply scale-out and when to switch between architectures. *scc*'s distinction between remote persistent datasets and local transient datasets enables it to capture the costs associated with data redistribution.

Network configuration and CPU diversity. *scc*'s specification of application behavior can be used to infer the communication pattern among the application's components, and thus inform configuration of the cluster's network. For example, in the case of ProductSearch, *scc* can infer from the application specification that the workers communicate only with the HDFS repository but not among themselves. *scc* can then use this information to recommend a bi-partite network with servers hosting HDFS on one side and servers hosting workers on the other side. *scc* can also be readily extended to choose among a range of CPUs; the application specification simply needs to include for every compute operation the time required for that operation on each type of CPU.

7 Related work

Our work builds upon and shares some similarities with several lines of prior work.

Tuning storage: Minerva [5], Hippodrome [7], and Rome [34] automate the provisioning of disk arrays with a similar approach of characterizing workloads and storage. Ursa Minor [2] varies erasure coding parameters depending on an application's availability requirements. PADS [9] is configurable to build a wide range of replication systems with varying consistency semantics. In contrast to all of these efforts, we consider an application's storage and compute requirements in unison. Moreover, we choose among different storage media such as disk, SSD, and DRAM to minimize cost, with multiple media possibly being used for the same application.

Application modeling: Bodik et al. [10] infer application performance models by applying machine learning techniques on statistics gathered by monitoring the application execution. Thereska et al. [31] predict performance across application configurations based on statistical models. IRONModel [32] corrects deviations between the performance of running systems and high fidelity models. In all cases, since application models are tuned to specific cluster configurations, they are not directly applicable to alternative hardware configurations.

Stewart and Shen [30] build performance models of multi-component applications to aid in the placement of application components on a given cluster. Osogami and Itoko [25] apply hill-climbing techniques to automatically determine web-server parameters, and Liu et al. [20] construct a queuing model for a three-tiered web service to predict throughput and response times. Again, all of these consider a fixed hardware configuration.

Application-specific cluster architectures: Application developers have converged on a range of cluster architectures for individual applications. Several web services employ DRAM caches using distributed in-memory storage systems [21, 26]. Applications such as WER [16] use clusters that have separate sets of machines for compute and storage. FAWN [6] and Gordon [11] use SSDs to build performant yet power-efficient distributed data processing systems. MR-Perf [33] and Starfish [18] use an approach similar to *scc* but focus solely on predicting cluster requirements of MapReduce setups. *scc* not only infers these cost-effective architectures for existing applications, but also enables the inference of the right cluster architecture for emerging applications.

Storage and computing services. There been a few recent attempts [19, 15] at satisfying SLAs in the setting of a compute and storage cluster shared across applications. Such multi-application environments have also seen the recent emergence of virtual storage appliances. *scc* is targeted at the still significantly more common scenario of cluster deployments for a single application.

8 Conclusions

The thesis of our work is that deployment of applications on clusters is more cost-effective if informed by characterizations of application behavior and hardware properties. Towards this end, we presented how these inputs can be specified, and we developed *scc* to compile these inputs into cost-effective cluster configurations. Our experiments in applying *scc* to a range of application workloads and storage options show that *scc* captures sufficient detail to prescribe the right combination of storage and server hardware at the right scale; modifying the architecture or reducing the scale leads to significant performance degradation. To meet application demands, *scc* often predicts heterogeneous cluster architectures that result in significant cost savings compared to simply scaling out homogeneous architectures. We plan to apply *scc* to other popular applications to determine more fine-grained characteristics from which it could benefit, and use *scc*'s application specification to select appropriate CPUs and optimize network costs. We also plan to develop tools to make it easier to put together hardware and application specifications.

Acknowledgments

We thank the anonymous reviewers and our shepherd Lakshmi Bairavasundaram for their feedback on previous versions of this paper. We also thank Brian Cooper, Khaled Elmeleegy, Garth Goodson, and Kaladhar Voruganti for their input at various stages of this project. This research was partially supported by a NetApp Faculty Fellowship.

References

[1] Amazon EC2 instance types. `http://aws.amazon.com/ec2/instance-types`.

[2] M. Abd-El-Malek, W. V. Courtright II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa minor: Versatile cluster-based storage. In *FAST*, 2005.

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

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

[5] G. A. Alvarez, E. Borowsky, S. Go, T. H. Romer, R. A. Becker-Szendy, R. A. Golding, A. Merchant, M. Spasojevic, A. C. Veitch, and J. Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems*, 2001.

[6] D. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *SOSP*, 2009.

[7] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. C. Veitch. Hippodrome: Running circles around storage administration. In *FAST*, 2002.

[8] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *OSDI*, 2004.

[9] N. Belaramani, J. Zheng, A. Nayate, R. Soul, M. Dahlin, and R. Grimm. PADS: A policy architecture for data replication systems. In *NSDI*, 2009.

[10] P. Bodik, R. Griffith, C. Sutton, A. Fox, M. I. Jordan, and D. A. Patterson. Automatic exploration of datacenter performance regimes. In *Proc. of the 1st workshop on Automated control for datacenters and clouds*, 2009.

[11] A. M. Caulfield, L. M. Grupp, and S. Swanson. Gordon: Using flash memory to build fast, power-efficient clusters for data-intensive applications. In *ASPLOS*, 2009.

[12] C. Chekuri and S. Khanna. On multi-dimensional packing problems. *SIAM Journal on Computing*, 2004.

[13] Cloudera. Cloudera´s support team shares some basic hardware recommendations. `http://www.cloudera.com/blog/2010/03/`.

[14] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, D. Burger, B. Lee, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *SOSP*, 2009.

[15] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.

[16] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: Ten years of implementation and experience. In *SOSP*, 2009.

[17] Google Merchant Center. `http://www.google.com/merchants`.

[18] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of MapReduce programs. In *VLDB*, 2011.

[19] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.

[20] X. Liu, J. Heo, and L. Sha. Modeling 3-tiered web applications. In *MASCOTS*, 2005.

[21] Memcached. `http://memcached.org`.

[22] D. A. Menascé, V. A. F. Almeida, and L. W. Dowdy. *Capacity planning and performance modeling: from mainframes to client-server systems*. Prentice-Hall, Inc., 1994.

[23] NetApp Inc. Microsoft Exchange 2007 deployment for 2,000 to 5,000 users integrated with high availability, backups, and disaster recovery. `http://media.netapp.com/documents/ra-0001-0509.pdf`.

[24] NetApp Inc. Oracle database dev/test reference architecture using data guard and SnapManager for Oracle deployment guide. `http://media.netapp.com/documents/ra-0002.pdf`.

[25] T. Osogami and T. Itoko. Finding probably better system configurations quickly. In *SIGMETRICS*, 2006.

[26] J. K. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for RAMClouds: Scalable high-performance storage entirely in DRAM. *SIGOPS OSR*, 2009.

[27] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan. Measurement and analysis of TCP throughput collapse in cluster-based storage systems. In *FAST*, 2008.

[28] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 1992.

[29] Sort benchmark home page. `http://sortbenchmark.org/`.

[30] C. Stewart and K. Shen. Performance modeling and system management for multi-component online services. In *NSDI*, 2005.

[31] E. Thereska, B. Doebel, A. X. Zheng, and P. Nobel. Practical performance models for complex, popular applications. In *SIGMETRICS*, 2010.

[32] E. Thereska and G. R. Ganger. IRONModel: Robust performance models in the wild. In *SIGMETRICS*, 2008.

[33] G. Wang, A. R. Butt, P. Pandey, and K. Gupta. A simulation approach to evaluating design decisions in mapreduce setups. In *MASCOTS*, 2009.

[34] J. Wilkes. Traveling to Rome: QoS specifications for automated storage system management. In *IWQoS*, 2001.

iDedup: Latency-aware, inline data deduplication for primary storage

Kiran Srinivasan, Tim Bisson, Garth Goodson, Kaladhar Voruganti

NetApp, Inc.
{skiran, tbisson, goodson, kaladhar}@netapp.com

Abstract

Deduplication technologies are increasingly being deployed to reduce cost and increase space-efficiency in corporate data centers. However, prior research has not applied deduplication techniques inline to the request path for latency sensitive, primary workloads. This is primarily due to the extra latency these techniques introduce. Inherently, deduplicating data on disk causes fragmentation that increases seeks for subsequent sequential reads of the same data, thus, increasing latency. In addition, deduplicating data requires extra disk IOs to access on-disk deduplication metadata. In this paper, we propose an inline deduplication solution, iDedup, for primary workloads, while minimizing extra IOs and seeks.

Our algorithm is based on two key insights from real-world workloads: i) spatial locality exists in duplicated primary data; and ii) temporal locality exists in the access patterns of duplicated data. Using the first insight, we selectively deduplicate only sequences of disk blocks. This reduces fragmentation and amortizes the seeks caused by deduplication. The second insight allows us to replace the expensive, on-disk, deduplication metadata with a smaller, in-memory cache. These techniques enable us to tradeoff capacity savings for performance, as demonstrated in our evaluation with real-world workloads. Our evaluation shows that iDedup achieves 60-70% of the maximum deduplication with less than a 5% CPU overhead and a 2-4% latency impact.

1 Introduction

Storage continues to grow at an explosive rate of over 52% per year [10]. In 2011, the amount of data will surpass 1.8 zettabytes [17]. According to the IDC [10], to reduce costs and increase storage efficiency, more than 80% of corporations are exploring deduplication technologies. However, there is a huge gap in the current capabilities of deduplication technology. No deduplication systems exist that deduplicate *inline* with client requests for latency sensitive primary workloads. All prior deduplication work focuses on either: i) throughput sensitive archival and backup systems [8, 9, 15, 21, 26, 39, 41]; or ii) latency sensitive primary systems that deduplicate data *offline* during idle time [1, 11, 16]; or iii) file systems with inline deduplication, but agnostic to performance [3, 36]. This paper introduces two novel insights that enable latency-aware, *inline*, primary deduplication.

Many primary storage workloads (e.g., email, user directories, databases) are currently unable to leverage the benefits of deduplication, due to the associated latency costs. Since offline deduplication systems impact latency the least, they are currently the best option; however, they are inefficient. For example, offline systems require additional storage capacity to absorb the writes prior to deduplication, and excess disk bandwidth to perform reads and writes during deduplication. This additional disk bandwidth can impact foreground workloads. Additionally, inline compression techniques also exist [5, 6, 22, 38] that are complementary to our work.

The challenge of inline deduplication is to not increase the latency of the already latency sensitive, foreground operations. Reads are affected by the fragmentation in data layout that naturally occurs when deduplicating blocks across many disks. As a result, subsequent sequential reads of deduplicated data are transformed into random IOs resulting in significant seek penalties. Most of the deduplication work occurs in the write path; i.e., generating block hashes and finding duplicate blocks. To identify duplicates, on-disk data structures are accessed. This leads to extra IOs and increased latency in the write path. To address these performance concerns, it is necessary to minimize any latencies introduced in both the read and write paths.

We started with the realization that in order to improve latency a tradeoff must be made elsewhere. Thus, we were motivated by the question: *Is there a tradeoff between performance and the degree of achievable dedu-*

plication? While examining real-world traces [20], we developed two key insights that ultimately led to an answer: i) spatial locality exists in the duplicated data; and ii) temporal locality exists in the accesses of duplicated data. The first observation allows us to amortize the seeks caused by deduplication by only performing deduplication when a sequence of on-disk blocks are duplicated. The second observation enables us to maintain an in-memory fingerprint cache to detect duplicates in lieu of any on-disk structures. The first observation mitigates fragmentation and addresses the extra read path latency; whereas, the second one removes extra IOs and lowers write path latency. These observations lead to two control parameters: i) the minimum number of sequential duplicate blocks on which to perform deduplication; and ii) the size of the in-memory fingerprint cache. By adjusting these parameters, a tradeoff is made between the capacity savings of deduplication and the performance impact to the foreground workload.

This paper describes the design, implementation and evaluation of our deduplication system (iDedup) built to exploit the spatial and temporal localities of duplicate data in primary workloads. Our evaluation shows that good capacity savings are achievable (between 60%-70% of maximum) with a small impact to latency (2-4% on average). In summary, our key contributions include:

- Insights on spatial and temporal locality of duplicated data in real-world, primary workloads.
- Design of an inline deduplication algorithm that leverages both spatial and temporal locality.
- Implementation of our deduplication algorithm in an enterprise-class, network attached storage system.
- Implementation of efficient data structures to reduce resource overheads and improve cacheability.
- Demonstration of a viable tradeoff between performance and capacity savings via deduplication.
- Evaluation of our algorithm using data from real-world, production, enterprise file system traces.

The remainder of the paper is as follows: Section 2 provides background and motivation of the work; Section 3 describes the design of our deduplication system; Section 4 describes the system's implementation; Section 5 evaluates the implementation; Section 6 describes related work, and Section 7 concludes.

2 Background and motivation

Thus far, the majority of deduplication research has targeted improving deduplication within the backup and archival (or secondary storage) realm. As shown in Table 1, very few systems provide deduplication for latency sensitive primary workloads. We believe that this is due to the significant challenges in performing deduplication without affecting latency, rather than the lack of benefit deduplication provides for primary workloads. Our system is specifically targeted at this gap.

Type	**Offline**	**Inline**
Primary, latency sensitive	NetApp ASIS [1], EMC Celerra [11], StorageTank [16],	***i*Dedup** (This paper)
Secondary, throughput sensitive	(No motivation for systems in this category)	EMC DDFS [41], EMC Cluster [8] DeepStore [40], NEC HydraStor [9], Venti [31], SiLo [39], Sparse Indexing [21], ChunkStash [7], Foundation [32], Symantec [15], EMC Centera [24], GreenBytes [13]

Table 1: *Table of related work:.* The table shows how this paper, *iDedup*, is positioned relative to some other relevant work. Some primary, inline deduplication file systems (like ZFS [3]) are omitted, since they are not optimized for latency.

The remainder of this section further describes the differences between primary and secondary deduplication systems and describes the unique challenges faced by primary deduplication systems.

2.1 Classifying deduplication systems

Although many classifications for deduplication systems exist, they are usually based on internal implementation details, such as the fingerprinting (hashing) scheme or whether fixed sized or variable sized blocks are used. Although important, these schemes are usually orthogonal to the types of workloads their system supports. Similar to other storage systems, deduplication systems can be broadly classified as *primary* or *secondary* depending on the workloads they serve. Primary systems are used for primary workloads. These workloads tend to be latency sensitive and use RPC based protocols, such as NFS [30], CIFS [37] or iSCSI [35]. On the other hand, secondary systems are used for archival or backup purposes. These workloads process large amounts of data, are throughput sensitive and are based on streaming protocols.

Primary and secondary deduplication systems can be further subdivided into *inline* and *offline* deduplication systems. Inline systems deduplicate requests in the write path before the data is written to disk. Since inline deduplication introduces work into the critical write path, it often leads to an increase in request latency. On the other hand, offline systems tend to wait for system idle time to deduplicate previously written data. Since no operations are introduced within the write path; write latency is not affected, but reads remain fragmented.

Content addressable storage (CAS) systems (e.g., [24, 31]) naturally perform inline deduplication, since blocks are typically addressed by their fingerprints. Both archival and CAS systems are sometimes used for primary storage. Likewise, a few file systems that perform inline deduplication (e.g., ZFS [3] and SDFS [36]) are also used for primary storage. However, none of these systems are specifically optimized for latency sensitive workloads while performing inline deduplication. Their design for maximum deduplication introduces extra IOs and does not address fragmentation.

Primary inline deduplication systems have the following advantages over offline systems:

1. Storage provisioning is easier and more efficient: Offline systems require additional space to absorb the writes prior to deduplication processing. This causes a temporary bloat in storage usage leading to inaccurate space accounting and provisioning.
2. No dependence on system idle time: Offline systems use idle time to perform deduplication without impacting foreground requests. This is problematic when the system is busy for long periods of time.
3. Disk-bandwidth utilization is lower: Offline systems use extra disk bandwidth when reading in the staged data to perform deduplication and then again to write out the results. This limits the total bandwidth available to the system.

For good reason, the majority of prior deduplication work has focused on the design of inline, secondary deduplication systems. Backup and archival workloads typically have a large amount of duplicate data, thus the benefit of deduplication is large. For example, reports of 90+% deduplication ratios are not uncommon for backup workloads [41], compared to the 20-30% we observe from our traces of primary workloads. Also, since backup workloads are not latency sensitive, they are tolerant to delays introduced in the request path.

2.2 Challenges of primary deduplication

The almost exclusive focus on maximum deduplication at the expense of performance has left a gap for latency sensitive workloads. Since primary storage is usually the most expensive, any savings obtained in primary systems has high cost advantages. Due to their higher cost ($/GB), deduplication is even more critical for flash based systems; nothing precludes our techniques from working with these systems. In order for primary, inline deduplication to be practical for enterprise systems, a number of challenges must be overcome:

- Write path: The metadata management and IO required to perform deduplication inline with the write request increases write latency.

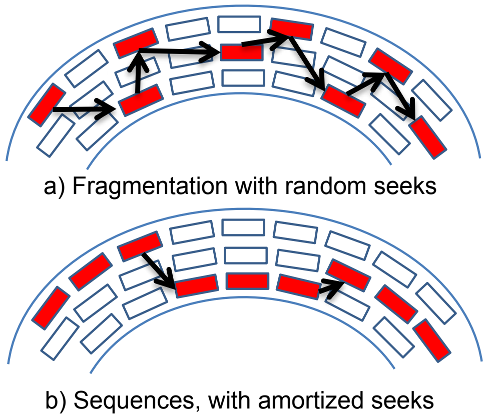

a) Fragmentation with random seeks
b) Sequences, with amortized seeks

Figure 1: *a) Increase in seeks due to increased fragmentation. b) The amortization of seeks using sequences.* This figure shows the amortization of seeks between disk tracks by using sequences of blocks (threshold=3).

- Read path: The fragmentation of otherwise sequential writes increases the number of disk seeks required during reads. This increases read latency.
- Delete path: The requirement to check whether a block can be safely deleted increases delete latency.

All of these penalties, due to deduplication, impact the performance of foreground workloads. Thus, primary deduplication systems only employ offline techniques to avoid interfering with foreground requests [1, 11, 16].

Write path: For inline, primary deduplication, write requests deduplicate data blocks prior to writing those blocks to stable storage. At a minimum, this involves fingerprinting the data block and comparing its signature within a table of previously written blocks. If a match is found, the metadata for the block, e.g., the file's block pointer, is updated to point to the existing block and no write to stable storage is required. Additionally, a reference count on the existing block is incremented. If a match is not found, the block is written to stable storage and the table of existing blocks is updated with the new block's signature and its storage location. The additional work performed during write path deduplication can be summarized as follows:

- Fingerprinting data consumes extra CPU resources.
- Performing fingerprint table lookups and managing the table persistently on disk requires extra IOs.
- Updating a block's reference count requires an update to persistent storage.

As one can see, the management of deduplication metadata, in memory, and on persistent storage, accounts for the majority of write path overheads. Even though much previous work has explored optimizing metadata management for inline, secondary systems (e.g., [2, 15, 21, 39, 41]), we feel that it is necessary to minimize all extra IO in the critical path for latency sensitive workloads.

Read path: Deduplication naturally fragments data that would otherwise be written sequentially. Fragmentation occurs because a newly written block may be deduplicated to an existing block that resides elsewhere on storage. Indeed, the higher the deduplication ratio, the higher the likelihood of fragmentation. Figure 1(a) shows the potential impact of fragmentation on reads in terms of the increased number seeks. When using disk based storage, the extra seeks can cause a substantial increase in read latency. Deduplication can convert sequential reads from the application into random reads from storage.

Delete path: Typically, some metadata records the usage of shared blocks. For example, a table of reference counts can be maintained. This metadata must be queried and updated inline to the deletion request. These actions can increase the latency of delete operations.

3 Design

In this section, we present the rationale that led to our solution, the design of our architecture, and the key design challenges of our deduplication system (iDedup).

3.1 Rationale for solution

To better understand the challenges of inline deduplication, we performed data analysis on real-world enterprise workloads [20]. First, we ran simulations varying the block size to see its effect on deduplication. We observed that the drop in deduplication ratio was less than linear with increasing block size. This implies duplicated data is clustered, thus indicating *spatial locality* in the data. Second, we ran simulations varying the fingerprint table size to determine if the same data is written repeatedly close in time. Again, we observed the drop in deduplication ratio was less than linear with decreasing table size. This implies duplicated data exhibits notable *temporal locality*, thus making the fingerprint table amenable to caching. Unfortunately, we could not test our hypothesis on other workloads due to the lack of traces with data duplication patterns.

3.2 Solution overview

We use the observations of spatial and temporal locality to derive an inline deduplication solution.

Spatial locality: We leverage the spatial locality to perform selective deduplication, thereby mitigating the extra seeks introduced by deduplication for sequentially read files. To accomplish this, we examine blocks at write time and attempt to only deduplicate full sequences of file blocks *if and only if the sequence of blocks are i) sequential in the file and ii) have duplicates that are sequential on disk.* Even with this optimization, sequential reads can still incur seeks between sequences. However, if we enforce an appropriate *minimum sequence length* for such sequences (the *threshold*), the extra seek cost is expected to be amortized; as shown by Figure 1(b). The threshold is a configurable parameter in our system. While some schemes employ a larger block size to leverage spatial locality, they are limited as the block size represents both the minimum and the maximum sequence length. Whereas, our threshold represents the minimum sequence length and the maximum sequence length is only limited by the file's size.

Inherently, due to our selective approach, only a subset of blocks are deduplicated, leading to lower capacity savings. Therefore, our inline deduplication technique exposes a tradeoff between capacity savings and performance, which we observe via experiments to be reasonable for certain latency sensitive workloads. For an optimal tradeoff, the threshold must be derived empirically to match the randomness in the workload. Additionally, to recover the lost savings, our system does not preclude executing other offline techniques.

Temporal locality: In all deduplication systems, there is a structure that maps the fingerprint of a block and its location on disk. We call this the deduplication metadata structure (or dedup-metadata for short). Its size is proportional to the number of blocks and it is typically stored on disk. Other systems use this structure as a lookup table to detect duplicates in the write path; this leads to extra, expensive, latency-inducing, random IOs.

We leverage the temporal locality by maintaining dedup-metadata as a *completely memory-resident, LRU cache*, thereby, avoiding extra dedup-metadata IOs. There are a few downsides to using a smaller, in-memory cache. Since we only cache mappings for a subset of blocks, we might not deduplicate certain blocks due to lack of information. In addition, the memory used by the cache reduces the file system's buffer cache size. This can lead to a lower buffer cache hit rate, affecting latency. On the other hand, the buffer cache becomes more effective by caching deduplicated blocks [19]. These observations expose another tradeoff between performance (hit rate) and capacity savings (dedup-metadata size).

3.3 Architecture

In this subsection, we provide an overview of our architecture. In addition, we describe the changes to the IO path to perform inline deduplication.

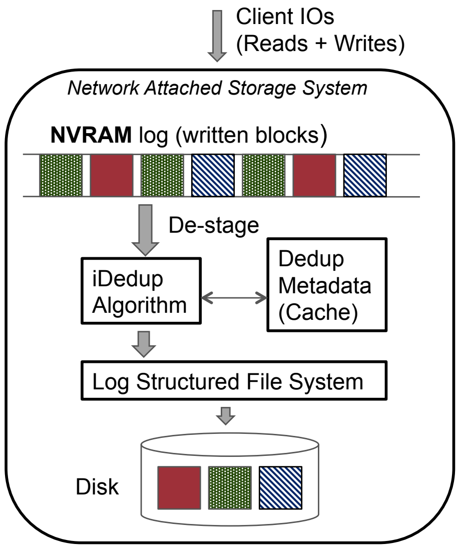

Client IOs
(Reads + Writes)
Network Attached Storage System
NVRAM log (written blocks)
De-stage
iDedup Algorithm
Dedup Metadata (Cache)
Log Structured File System
Disk

Figure 2: *iDedup Architecture.* Non-deduplicated blocks (different patterns) in NVRAM buffer are deduplicated by the iDedup algorithm before writing them to disk via the file system.

3.3.1 Storage system overview

An enterprise-class network-attached storage (NAS) system (as illustrated in Figure 2) is used as the reference system to build iDedup. For primary workloads, the system supports the NFS [30] and CIFS [37] RPC-based protocols. As seen in Figure 2, the system uses a log-structured file system [34] combined with non-volatile RAM (NVRAM) to buffer client writes to reduce response latency. These writes are periodically flushed to disk during the *destage* phase. Allocation of new disk blocks occur during this phase and is performed successively for each file written. Individual disk blocks are identified by their unique *disk block numbers* (DBNs). File metadata, containing the DBNs of its blocks, is stored within an *inode* structure. Given our objective to perform inline deduplication, the newly written (*dirty*) blocks need to be deduplicated during the destage phase. By performing deduplication during destage, the system benefits by not deduplicating short-lived data that is overwritten or deleted while buffered in NVRAM. Adding inline deduplication modifies the write path significantly.

3.3.2 Write path flow

Compared to the normal file system write path, we add an extra layer of deduplication processing. As this layer consumes extra CPU cycles, it can prolong the total time required to allocate dirty blocks and affect time-sensitive file system operations. Moreover, any extra IOs in this layer can interfere with foreground read requests. Thus, this layer must be optimized to minimize overheads. On the other hand, there is an opportunity to overlap deduplication processing with disk write IOs in the destage phase. The following steps take place in the write path:

1. For each file, the list of dirty blocks is obtained.
2. For each dirty block, we compute its fingerprint (hash of the block's content) and perform a lookup in the dedup-metadata structure using the hash as the key.
3. If a duplicate is found, we examine adjacent blocks, using the iDedup algorithm (Section 3.4), to determine if it is part of a duplicate sequence.
4. While examining subsequent blocks, some duplicate sequences might end. In those cases, the length of the sequence is determined, if it is greater than the configured threshold, we mark the sequence for deduplication. Otherwise, we allocate new disk blocks and add the fingerprint metadata for these blocks.
5. When a duplicate sequence is found, the DBN of each block in the sequence is obtained and the file's metadata is updated and eventually written to disk.
6. Finally, to maintain file system integrity in the face of deletes, we update reference counts of the duplicated blocks in a separate structure on disk.

3.3.3 Read path flow

Since iDedup updates the file's metadata as soon as deduplication occurs, the file system cannot distinguish between a duplicated block and a non-duplicated one. This allows file reads to occur in the same manner for all files, regardless of whether they contain deduplicated blocks. Although sequential reads may incur extra seeks due to deduplication, having a minimum sequence length helps amortize this cost. Moreover, if we pick the threshold closer to the expected sequentiality of a workload, then the effects of those seeks can be hidden.

3.3.4 Delete path flow

As mentioned in the write path flow, deduplicated blocks need to be reference counted. During deletion, the reference count of deleted blocks is decremented and only blocks with no references are freed. In addition to updating the reference counts, we also update the in-memory dedup-metadata when a block is deleted.

3.4 iDedup algorithm

The iDedup deduplication algorithm has the following key design objectives:

1. The algorithm should be able to identify sequences of file blocks that are duplicates and whose corresponding DBNs are sequential.
2. The largest duplicate sequence for a given set of file blocks should be identified.
3. The algorithm should minimize searches in the dedup-metadata to reduce CPU overheads.

4. The algorithm execution should overlap with disk IO during the destage phase and not prolong the phase.
5. The memory and CPU overheads caused by the algorithm should not prevent other file system processes from accomplishing their tasks in a timely manner.
6. The dedup-metadata must be optimized for lookups.

More details of the algorithm are presented in Section 4.2. Next, we describe the design elements that enable these objectives.

3.4.1 Dedup-metadata cache design

The dedup-metadata is maintained as a cache with one entry per block. Each entry maps the fingerprint of a block to its DBN on disk. We use LRU as the cache replacement policy; other replacement policies did not perform better than the simpler LRU scheme.

The choice of the fingerprint influences the size of the entry and the number of CPU cycles required to compute it. By leveraging processor hardware assists (for e.g., Intel AES [14]) to compute stronger fingerprints (like SHA-2, SHA-256 [28], etc.), the CPU overhead can be greatly mitigated. However, longer, 256-bit fingerprints increase the size of each entry. In addition, a DBN of 32-bits to 64-bits must also be kept within the entry, thus making the minimum entry size 36 bytes. Given a block size of 4 KB (typical of many file systems), the cache entries comprise an overhead of 0.8% of the total size. Since we keep the cache in memory, this overhead is significant as it reduces the number of cached blocks.

In many storage systems, memory not reserved for data structures is used by the buffer cache. Hence, the memory used by the dedup-metadata cache comes at the expense of a larger buffer cache. Therefore, the effect of the dedup-metadata cache on the buffer cache hit ratio needs to be evaluated empirically to size the cache.

3.4.2 Duplicate sequence processing

This subsection describes some common design issues in duplicate sequence identification.

Sequence identification: The goal is to identify the largest sequence among the list of potential sequences. This can be done in multiple ways:

- Breadth-first: Start by scanning blocks in order; concurrently track all possible sequences; and decide on the largest when a sequence terminates.
- Depth-first: Start with a sequence and pursue it across the blocks until it terminates; make multiple passes until all sequences are probed; and then pick the largest. Information gathered during one pass can be utilized to make subsequent passes more efficient.

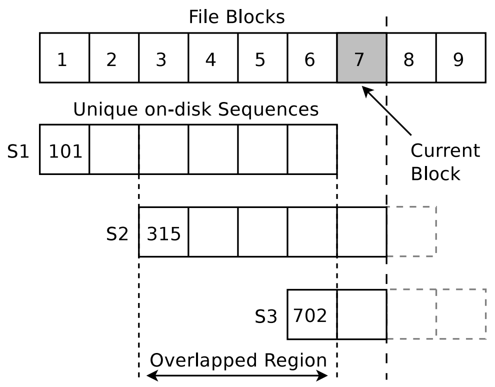


Figure 3: *Overlapped sequences.* This figure shows an example of how the algorithm works with overlapped sequences.

In practice, we observed long chains of blocks during processing (order of 1000s). Since multiple passes is too expensive, we use the breadth-first approach.

Overlapped sequences: Choosing between a set of overlapped sequences can prove problematic. An example of how overlapping sequences are handled is illustrated in Figure 3. Assume a threshold of 4. Scanning from left to right, multiple sequences match the set of file blocks. As we process the 7th block, one of the sequences terminates (S1) with a length 6. But, sequences S2 and S3 have not yet terminated and have blocks overlapping with S1. Since S1 is longer than the threshold (4), we can deduplicate the file blocks matching those in S1. However, by accepting S1, we are rejecting the overlapped blocks from S2 or S3; this is the dilemma. It is possible that either S2 or S3 could potentially lead to a longer sequence going forward, but it is necessary to make a decision about S1. Since it is not possible to know the best outcome, we use the following heuristic: we determine if the set of non-overlapped blocks is greater than threshold, if so, we deduplicate them. Otherwise, we defer to the unterminated sequences, as they may grow longer. Thus, in the example, we reject S1 for this reason.

3.4.3 Threshold determination

The minimum sequence threshold is a workload property that can only be derived empirically. The ideal threshold is one that most closely matches the workload's natural sequentiality. For workloads with more random IO, it is possible to set a lower threshold because deduplication should not worsen the fragmentation. It is possible to have a real-time, adaptive scheme that sets the threshold based on the randomness of the workload. Although valuable, this investigation is beyond this paper's scope.

4 Implementation

In this section, we present the implementation and optimizations of our inline deduplication system. The implementation consists of two subsystems: i) the dedup-metadata management; and ii) the iDedup algorithm.

4.1 Dedup-metadata management

The dedup-metadata management subsystem is comprised of several components:

1. Dedup-metadata cache (in RAM): Contains a pool of block entries (*content-nodes*) that contain deduplication metadata organized as a cache.
2. Fingerprint hash table (in RAM): This table maps a fingerprint to DBN(s).
3. DBN hash table (in RAM): This table maps a DBN to its content-node; used to delete a block.
4. Reference count file (on disk): Maintains reference counts of deduplicated file system blocks in a file.

We explore each of them next.

4.1.1 Dedup-metadata cache

This is a fixed-size pool of small entries called content-nodes, managed as an LRU cache. The size of this pool is configurable at compile time. Each content-node represents a single disk block and is about 64 bytes in size. The content-node contains the block's DBN (a 4 B integer) and its fingerprint. In our prototype, we use the MD5 checksum (128-bit) [33] of the block's contents as its fingerprint. Using a stronger fingerprint (like SHA-256) would increase the memory overhead of each entry by 25%, thus leading to fewer blocks cached. Other than this effect, using MD5 is not expected to alter other experimental results.

All the content-nodes are allocated as a single global array. This allows the nodes to be referenced by their array index (a 4 byte value) instead of by a pointer. This saves 4 bytes per pointer in 64-bit systems. Each content-node is indexed by three data structures: the fingerprint hash table, the DBN hash table and the LRU list. This adds two pointers per index (to doubly link the nodes in a list or tree), thus totaling six pointers per content-node. Therefore, by using array indices instead of pointers we save 24 bytes per entry (37.5%).

4.1.2 Fingerprint hash table

This hash table contains content-nodes indexed by their fingerprint. It enables a block's duplicates to be identified by using the block's fingerprint. As shown in Figure 4, each hash bucket contains a single pointer to the root of a red-black tree containing the collision list for that bucket. This is in contrast to a traditional hash table with a doubly linked list for collisions at the cost of two pointers per bucket. The red-black tree implementation is an optimized, left-leaning, red-black tree [12].

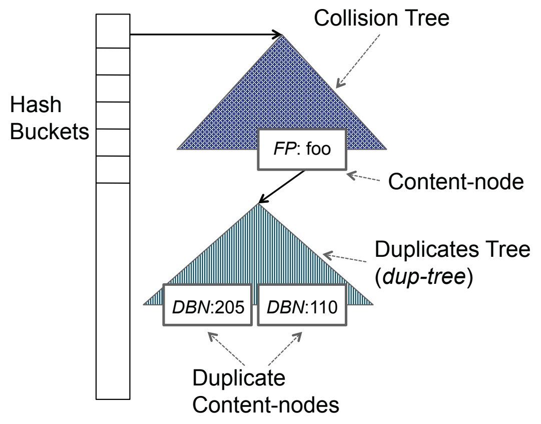


Figure 4: *Fingerprint Hash Table.* The fingerprint hash table with hash buckets as pointers to collision trees. Content-node with fingerprint 'foo' has duplicate content-nodes in a tree (dup-tree) with DBNs 205 and 110.

With uniform distribution, each hash bucket is designed to hold 16 entries, ensuring an upper-bound of 4 searches within the collision tree (tree search cost is O(logN)). By reducing the size of the pointers and the number of pointers per bucket, the per-bucket overhead is reduced, thus providing more buckets for the same memory size.

Each collision tree content-node represents a unique fingerprint value in the system. For thresholds greater than one, it is possible for multiple DBNs to have the same fingerprint, as they can belong to different duplicate sequences. Therefore, all the content-nodes that represent duplicates of a given fingerprint are added to another red-black tree, called the *dup-tree* (see Figure 4). This tree is rooted at the first content-node that maps to that fingerprint. There are advantages to organizing the duplicate content-nodes in a tree, as explained in the iDedup algorithm section (Section 4.2).

4.1.3 DBN hash table

This hash table indexes content-nodes by their DBNs. Its structure is similar to the fingerprint hash table without the dup-tree. It facilitates the deletion of content-nodes when the corresponding blocks are removed from the system. During deletion, blocks can only be identified by their DBNs (otherwise the data must be read and hashed). The DBN is used to locate the corresponding content-node and delete it from all dedup-metadata.

4.1.4 Reference count file

The *refcount file* stores the reference counts of all deduplicated blocks on disk. It is ordered by DBN and maintains a 32-bit counter per block. When a block is deleted, its entry in the refcount file is decremented. When the reference count reaches zero, the block's content-node is removed from all dedup-metadata and the block is

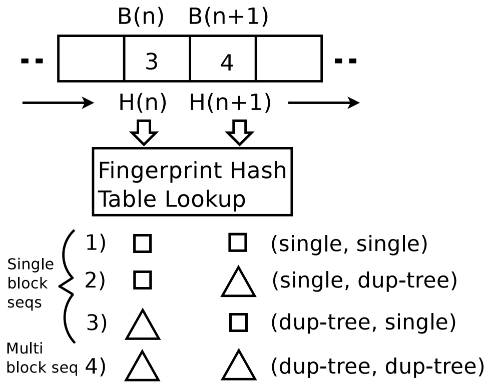


Figure 5: *Identification of sequences.* This figure shows how sequences are identified.

marked free in the file system's metadata. The refcount file is also updated when a block is written. By deduplicating sequential blocks, we observe that refcount updates are often collocated to the same disk blocks, thereby amortizing IOs to the refcount file.

4.2 iDedup algorithm

For each file, the iDedup algorithm has three phases:

1. Sequence identification: Identify duplicate block sequences for file blocks.
2. Sequence pruning: Process duplicate sequences based on their length.
3. Sequence deduplication: Deduplicate sequences greater than the configured threshold.

We examine these phases next.

4.2.1 Sequence identification

In this phase, a set of newly written blocks, for a particular file, are processed. We use the breadth-first approach for determining duplicate sequences. We start by scanning the blocks in order and utilize the fingerprint hash table to identify any duplicates for these blocks. We filter the blocks to pick only data blocks that are complete (i.e., of size 4 KB) and that do not belong to special or system files (e.g., the refcount file). During this pass, we also compute the MD5 hash for each block.

In Figure 5, the blocks B(n) (n = 1,2,3....) and the corresponding fingerprints H(n) (n = 1,2,3...) are shown. Here, *n* represents the block's offset within the file (the file block number or FBN). The minimum length of a duplicate sequence is two; so, we examine blocks in pairs; i.e., B(1) and B(2) first, B(2) and B(3) next and so on. For each pair, e.g., B(n) and B(n+1) (see Figure 5), we perform a lookup in the fingerprint hash table for H(n) and H(n+1), if neither of them is a match, we allocate the blocks on disk normally and move to the next pair. When we find a match, the matching content-nodes may have

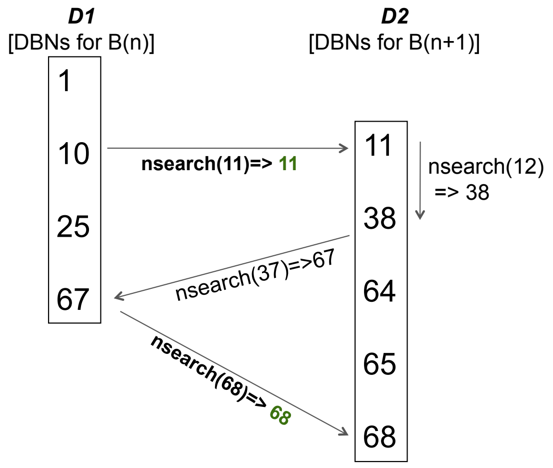


Figure 6: *Sequence identification example.* Sequence identification for blocks with multiple duplicates. D1 represents the dup-tree for block B(n) and D2 the dup-tree for B(n+1).

more than one duplicate (i.e., a dup-tree) or just a single duplicate (i.e., just an single DBN). Accordingly, to determine if a sequence exists across the pair, we have one of four conditions. They are listed below in increasing degrees of difficulty; they are also illustrated in Figure 5.

1. Both H(n) and H(n+1) match a single content-node: Simplest case, if the DBN of H(n) is b, and DBN of H(n+1) is (b+1), then we have a sequence.
2. H(n) matches a single content-node, H(n+1) matches a dup-tree content-node: If the DBN of H(n) is b; search for (b+1) in the dup-tree of H(n+1).
3. H(n) matches a dup-tree, H(n+1) matches a single content-node: Similar to the previous case with H(n) and H(n+1) swapped.
4. Both H(n) and H(n+1) match dup-tree content-nodes: This case is the most complex and can lead to multiple sequences. It is discussed in greater detail below.

When both H(n) and H(n+1) match entries with dup-trees, we need to identify all possible sequences that can start from these two blocks. The optimized red-black tree used for the dup-trees has a search primitive, `nsearch(x)`, that returns 'x' if 'x' is found; or the next largest number after 'x'; or error if 'x' is already the largest number. The cost of `nsearch` is the same as that of a regular tree search (O(log N)). We use this primitive to quickly search the dup-trees for all possible sequences. This is illustrated via an example in Figure 6.

In our example, we show the dup-trees as two sorted list of DBNs. First, we compute the minimum and maximum overlapping DBNs between the dup-trees (i.e., 10 and 68 in the figure), all sequences will be within this range. We start with 10, since this is in *D1*, the dup-tree of H(n). We then perform a `nsearch(11)` in *D2*, the dup-tree of H(n+1), which successfully leads to a sequence. Since the numbers are ordered, we perform a `nsearch(12)` in D2 to find the next largest potential sequence number; the result is 38. Next, to pair with 38, we perform `nsearch(37)` in D1. However, it results in 67 (not a sequence). Similarly, since we obtained 67 in D1, we perform `nsearch(68)` in D2, thus, yielding another sequence. In this fashion, with a minimal number of searches using the `nsearch` primitive, we are able to glean all possible sequences between the two blocks.

It is necessary to efficiently record, manage, and identify the sequences that are growing and those that have terminated. For each discovered sequence, we manage it via a *sequence entry*: the tuple ⟨Last FBN of sequence, Sequence Size, Last DBN of sequence⟩. Suppose, B(n) and B(n+1) have started a sequence with sequence entry S1. Upon examining B(n+1) and B(n+2), we find that S1 grows and a new sequence, S2, is created. In such a scenario, we want to quickly search for S1 and update its contents and create a new entry for S2. Therefore, we maintain the sequence entries in a hash table indexed by a combination of the tuple fields. In addition, as we process the blocks, to quickly determine terminated sequences, we keep two lists of sequence entries: one for sequences that include the current block and another for sequences of the previous block. After sequence identification for a block completes, if a sequence entry is not in the current block's list, then it has terminated.

4.2.2 Sequence pruning

Once we determine the sequences that have terminated, we process them according to their sizes. If a sequence is larger than the threshold, we check for overlapping blocks with non-terminated sequences using the heuristic mentioned in Section 3.4.2, and only deduplicate the non-overlapped blocks if they form a sequence greater than the threshold. For sequences shorter than the threshold, the non-overlapped blocks are allocated by assigning them to new blocks on disk.

4.2.3 Deduplication of blocks

For each deduplicated block, the file's metadata is updated with the original DBN at the appropriate FBN location. The appropriate block in the refcount file is retrieved (a potential disk IO) and the reference count of the original DBN is incremented. We expect the refcount updates to be amortized across the deduplication of multiple blocks for long sequences.

5 Experimental evaluation

In this section, we describe the goals of our evaluation followed by details and results of our experiments.

5.1 Evaluation objectives

Our goal is to show that a reasonable tradeoff exists between performance and deduplication ratio that can be exploited by iDedup for latency sensitive, primary workloads. In our system, the two major tunable parameters are: i) the minimum duplicate sequence threshold, and ii) the in-memory dedup-metadata cache size. Using these paramaters we evaluate the system by replaying traces from two real-world, enterprise workloads to examine:

1. Deduplication ratio vs. threshold: We expect a drop in deduplication rate as threshold increases.
2. Disk fragmentation profile vs. threshold: We expect the fragmentation to decrease as threshold increases.
3. Client read response time vs. threshold: We expect the client read response time characteristics to follow the disk fragmentation profile.
4. System CPU utilization vs. threshold: We expect the utilization to increase slightly with the threshold.
5. Buffer cache hit rate vs. dedup-metadata cache size: We expect the buffer cache hit ratio to decrease as the metadata cache size increases.

We describe these experiments and their results next.

5.2 Experimental setup

All evaluation is done using a NetApp® FAS 3070 storage system running Data ONTAP® 7.3 [27]. It consists of: 8 GB RAM; 512 MB NVRAM; 2 dual-core 1.8 GHz AMD CPUs; and 3 10K RPM 144 GB FC Seagate Cheetah 7 disk drives in a RAID-0 stripe. The trace replay client has a 16-core, Intel® Xeon® 2.2 GHz CPU with 16 GB RAM and is connected by a 1 Gb/s network link.

We use two, real-world, CIFS traces obtained from a production, primary storage system that was collected and made available by NetApp [20]. One trace contains Corporate departments' data (MS Office, MS Access, VM Images, etc.), called the *Corporate* trace; it contains 19,876,155 read requests (203.8 GB total read) and 3,968,452 write requests (80.3 GB total written). The other contains Engineering departments' data (user home dirs, source code, etc.), called the *Engineering* trace; it contains 23,818,465 read requests (192.1 GB total read) and 4,416,026 write requests (91.7 GB total written). Each trace represents $\approx$ 1.5 months of activity. They are replayed without altering their data duplication patterns.

We use three dedup-metadata cache sizes: 1 GB, 0.5 GB and 0.25 GB, that caches block mappings for approximately 100%, 50% and 25% of all blocks written in the trace respectively. For the threshold, we use reference values of 1, 2, 4, and 8. Larger thresholds produce insignificant deduplication savings to be feasible.

Two key comparison points are used in our evaluation:

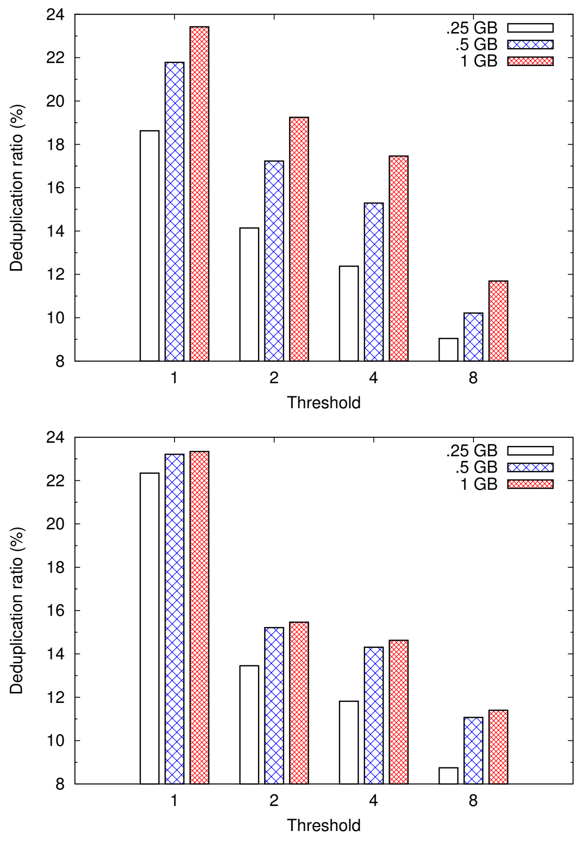


Figure 7: *Deduplication ratio vs. Threshold.* Deduplication ratio versus threshold for the different cache sizes for Corporate (top) and Engineering (bottom) traces.

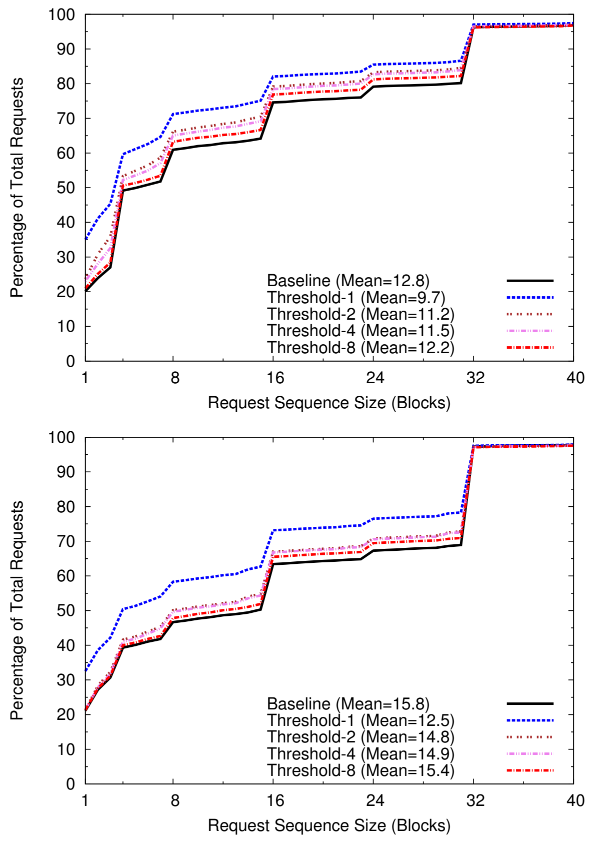


Figure 8: *Disk fragmentation profile.* CDF of number of sequential blocks in disk read requests for the Corporate (top) and Engineering (bottom) traces with a 1G cache.

1. The *Baseline* values represent the system without the iDedup algorithm enabled (i.e., no deduplication).
2. The Threshold-1 values represent the highest deduplication ratio for a given metadata cache size. Since a 1 GB cache caches all block mappings, Threshold-1 at 1 GB represents the maximum deduplication possible (with a 4 KB block size) and is equivalent to a static offline technique.

5.3 Deduplication ratio vs. threshold

Figure 7 shows the tradeoff in deduplication ratio (dedup-rate) versus threshold for both the workloads and different dedup-metadata sizes. For both the workloads, as the threshold increases, the number of duplicate sequences decrease, correspondingly the dedup-rate drops; there is a 50% decrease between Threshold-1 (24%) and 8 (13%), with a 1 GB cache. Our goal is to maximize the size of the threshold, while also maintaining a high dedup-rate. To evaluate this tradeoff, we look for a range of useful thresholds (> 1) where the drop in dedup-rate is not too steep; e.g., the dedup-rates between Threshold-2 and Threshold-4 are fairly flat. To minimize performance impact, we would pick the largest threshold that shows the smallest loss in dedup-rate: Threshold-4 from either graph. Moreover, we notice the drop in dedup-rate from Threshold-2 to Threshold-4 is same for 0.5 GB and 0.25 GB ($\approx$ 2%), showing a bigger percentage drop for smaller caches. For the Corporate workload, iDedup achieves a deduplication ratio between 66% (at Threshold-4, 0.25 GB) and 74% (at Threshold-4, 1 GB) of the maximum possible ($\approx$ 24% at Threshold-1, 1 GB). Similarly, with the Engineering workload, we achieve between 54% (at Threshold-4, 0.25 GB) and 62% (at Threshold-4, 1 GB) of the maximum ($\approx$ 23% at Threshold-1, 1 GB).

5.4 Disk fragmentation profile

To assess disk fragmentation due to deduplication, we gather the number of sequential blocks (request size) for each disk read request across all the disks and plot them as a CDF (cumulative distribution function). All CDFs are based on the average over three runs. Figure 8 shows the CDFs for both Corporate and Engineering workloads for a dedup-metadata cache of 1 GB. Other cache sizes show similar patterns. Since the request stream is the same for all thresholds, the difference in disk IO sizes, across the different thresholds, reflects the fragmentation of the file system's disk layout.

As expected, in both the CDFs, the Baseline shows the highest percentage of longer request sizes or sequentiality; i.e., the least fragmentation. Also, it can observed that the Threshold-1 line shows the highest amount of fragmentation. For example, there is a 11% increase in the number of requests smaller or equal to 8, between the Baseline and Threshold-1 for the Corporate workload and 12% for the Engineering workload. All the remaining thresholds (2, 4, 6, 8) show progressively less fragmentation, and have CDFs between the Baseline and the Threshold-1 line; e.g., a 2% difference between Baseline and Threshold-8 for the Corporate workload. Hence, to optimally choose a threshold, we suggest the tradeoff is made after empirically deriving the dedup-rate graph and the fragmentation profile. In the future, we envision enabling the system to automatically make this tradeoff.

5.5 Client response time behavior

Figure 9 (top graph) shows a CDF of client response times taken from the trace replay tool for varying thresholds of the Corporate trace at 1 GB cache size. We use response time as a measure of latency. For thresholds of 8 or larger, the behavior is almost identical to the Baseline (an average difference of 2% for Corporate and 4% for Engineering at Threshold 8) , while Threshold-2 and 4 (not shown) fall in between. We expect the client response time to reflect the fragmentation profile. However, the impact on client response time is lower due to the storage system's effective read prefetching.

As can be seen, there is a slowly shrinking gap between Threshold-1 and Baseline for larger response times ($>$ 2ms) comprising $\approx$ 10% of all requests. The increase in latency of these requests is due to the fragmentation effect and it affects the average response time. To quantify this better, we plot the difference between the two curves in the CDF (bottom graph of Figure 9) against the response time. The area under this curve shows the total contribution to latency due to the fragmentation effect. We find that it adds 13% to the average latency and a similar amount to the total runtime of the workload, which is significant. The Engineering workload has a similar pattern, although the effect is smaller (1.8% for average latency and total runtime).

5.6 System CPU utilization vs. threshold

We capture CPU utilization samples every 10 seconds from all the cores and compute the CDF for these values. Figure 10 shows the CDFs for our workloads with a 1 GB dedup-metadata cache. We expect Threshold-8 to consume more CPU because there are potentially more outstanding, unterminated sequences leading to more sequence processing and management. As expected, compared to the Baseline, the maximum difference in mean CPU utilization occurs at Threshold-8, but is relatively small: $\approx$ 2% for Corporate and $\approx$ 4% for Engineering. However, the CDFs for the thresholds exhibit a longer tail, implying a larger standard deviation compared to the Baseline, this is evident in the Engineering case but less so for Corporate. However, given that the change is small ($<$ 5%), we feel that the iDedup algorithm has little impact on the overall utilization. The results are similar across cache sizes, we chose the maximal 1 GB one, since that represents maximum work in sequence processing for the iDedup algorithm.

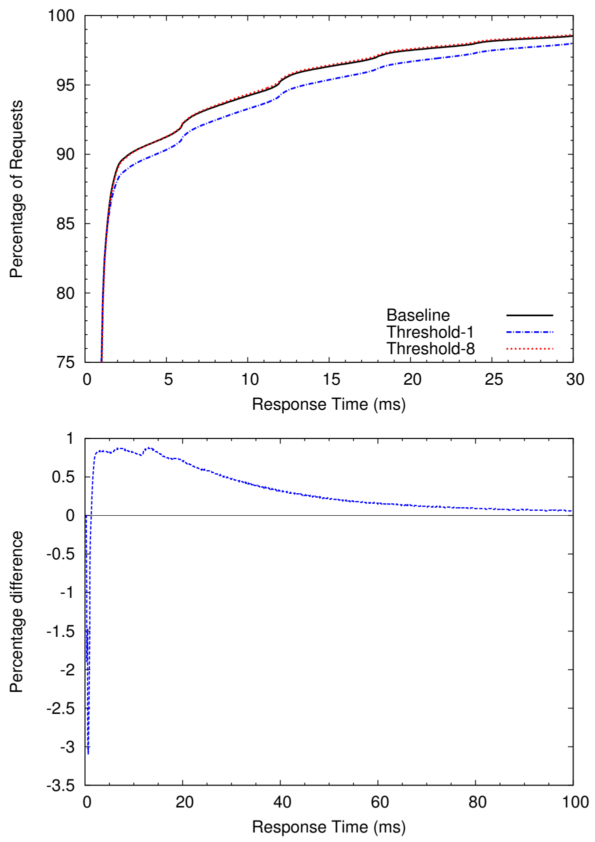
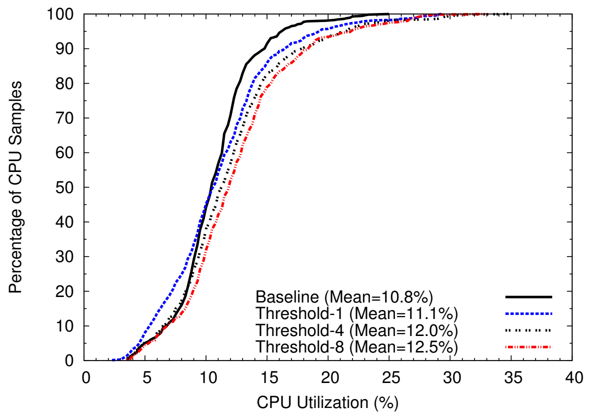


Figure 9: *Client response time CDF.* CDF of client response times for Corporate with a 1 GB cache (top); we highlight the region where the curves differ. The difference between the Baseline and Threshold of 1 CDFs (bottom).

5.7 Buffer cache hit ratio vs. metadata size

We observed the buffer cache hit ratio for different sizes of the dedup-metadata cache. The size of the dedup-metadata cache (and threshold) had no observable ef-



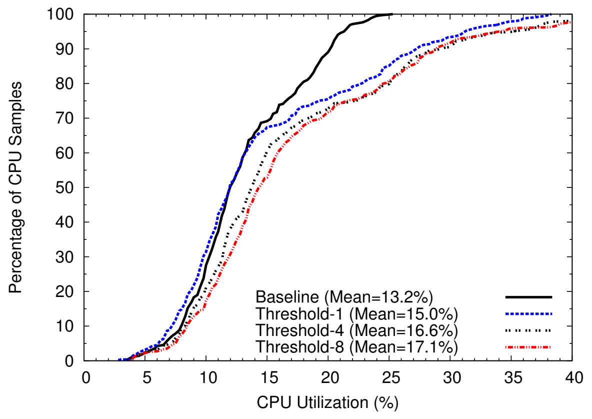


Figure 10: *CPU Utilization CDF.* CDF across all the cores for varying thresholds for Corporate (top) and Engineering (bottom) workloads with a 1 GB cache. Threshold-2 is omitted, since it almost fully overlaps Threshold-4.

fect on the buffer cache hit ratio for two reasons: i) the dedup-metadata cache size (max of 1 GB) is relatively small compared to the total memory (8 GB); and ii) the workloads' working sets fit within the buffer cache. The buffer cache hit ratio was steady for the Corporate (93%) and Engineering (96%) workloads. However, workloads with working sets that do not fit in the buffer cache would be impacted by the dedup-metadata cache.

6 Related work

Data storage efficiency can be realized via various complementary techniques such as thin-provisioning (not all of the storage is provisioned up front), data deduplication, and compression. As shown in Table 1 and as described in Section 2, deduplication systems can be classified as primary or secondary (backup/archival). Primary storage is usually optimized for IOPs and latency whereas secondary storage systems are optimized for throughput. These systems either process duplicates inline, at ingest time, or offline, during idle time.

Another key trade-off is with respect to the deduplication granularity. In file level deduplication (e.g., [18, 21, 40]), the potential gains are limited compared to deduplication at block level. Likewise, there are algorithms for fixed-sized block or variable-sized (e.g., [4, 23]) block deduplication. Finally, there are content addressable systems (CAS) that reference the object or block directly by its content hash; inherently deduplicating them [24, 31].

Although, we are unaware of any prior primary, inline deduplication systems, offline systems do exist. Some are block-based [1, 16], while others are file-based [11].

Complementary research has been done on inline compression for primary data [6, 22, 38]. Burrows et. al [5] describe an on-line compression technique for primary storage using a log-structured file system. In addition, offline compression products also exist [29].

The goals for inline secondary or backup deduplication systems are to provide high throughput and high deduplication ratio. Therefore, to reduce the amount of in-memory dedup-metadata footprint and the number of metadata IOs, various optimizations have been proposed [2, 15, 21, 39, 41]. Another inline technique, by Lillibridge et al. [21], leverages temporal locality with sampling to reduce dedup metadata size in the context of backup streams.

Deduplication systems have also leveraged flash storage to minimize the cost of metadata IOs [7, 25]. Clustered backup storage systems have been proposed for large datasets that cannot be managed by a single backup storage node [8].

7 Conclusion

In this paper, we describe iDedup, an inline deduplication system specifically targeting latency-sensitive, primary storage workloads. With latency sensitive workloads, inline deduplication has many challenges: fragmentation leading to extra disk seeks for reads, deduplication processing overheads in the critical path, and extra latency caused by IOs for dedup-metadata management.

To counter these challenges, we derived two insights by observing real-world, primary workloads: i) there is significant spatial locality on disk for duplicated data, and ii) temporal locality exists in the accesses of duplicated blocks. First, we leverage spatial locality to perform deduplication only when the duplicate blocks form long sequences on disk, thereby, avoiding fragmentation. Second, we leverage temporal locality by maintaining dedup-metadata in an in-memory cache to avoid extra IOs. From our evaluation, we see that iDedup offers significant deduplication with minimal resource overheads (CPU and memory). Furthermore, with careful threshold selection, a good compromise between performance and deduplication can be reached, thereby, making iDedup well suited to latency sensitive workloads.

References

[1] C. Alvarez. NetApp deduplication for FAS and V-Series deployment and implementation guide. Technical Report TR-3505, NetApp, 2011.

[2] D. Bhagwat, K. Eshghi, D. D. E. Long, and M. Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Proceedings of the 17th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '09)*, pages 1–9, Sept. 2009.

[3] J. Bonwick. Zfs deduplication. `http://blogs.oracle.com/bonwick/entry/zfs_dedup`, Nov. 2009.

[4] S. Brin, J. Davis, and H. Garcia-Molina. Copy detection mechanisms for digital documents. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 398–409, May 1995.

[5] M. Burrows, C. Jerian, B. Lampson, and T. Mann. On-line data compression in a log-structured file system. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 2–9, Boston, MA, Oct. 1992.

[6] C. Constantinescu, J. S. Glider, and D. D. Chambliss. Mixing deduplication and compression on active data sets. In *Proceedings of the 2011 Data Compression Conference*, pages 393–402, Mar. 2011.

[7] B. Debnath, S. Sengupta, and J. Li. Chunkstash: speeding up inline storage deduplication using flash memory. In *Proceedings of the 2010 USENIX Annual Technical Conference*, pages 16–16, June 2010.

[8] W. Dong, F. Douglis, K. Li, R. H. Patterson, S. Reddy, and P. Shilane. Tradeoffs in scalable data routing for deduplication clusters. In *Proceedings of the Ninth USENIX Conference on File and Storage Technologies (FAST)*, pages 15–29, Feb. 2011.

[9] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki. Hydrastor: A scalable secondary storage. In *Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FAST)*, pages 197–210, Feb. 2009.

[10] L. DuBois, M. Amaldas, and E. Sheppard. Key considerations as deduplication evolves into primary storage. White Paper 223310, Mar. 2011.

[11] EMC. Achieving storage efficiency through EMC Celerra data deduplication. White paper, Mar. 2010.

[12] J. Evans. Red-black tree implementation. `http://www.canonware.com/rb`, 2010.

[13] GreenBytes. GreenBytes, GB-X series. `http://www.getgreenbytes.com/products`.

[14] S. Gueron. Intel advanced encryption standard (AES) instructions set. White Paper, Intel, Jan. 2010.

[15] F. Guo and P. Efstathopoulos. Building a high-performance deduplication system. In *Proceedings of the 2011 USENIX Annual Technical Conference*, pages 25–25, June 2011.

[16] IBM Corporation. IBM white paper: IBM Storage Tank – A distributed storage system, Jan. 2002.

[17] IDC. The 2011 digital universe study. Technical report, June 2011.

[18] N. Jain, M. Dahlin, and R. Tewari. Taper: Tiered approach for eliminating redundancy in replica synchronization. In *Proceedings of the Fourth USENIX Conference on File and Storage Technologies (FAST)*, pages 281–294, Dec. 2005.

[19] R. Koller and R. Rangaswami. I/O deduplication: Utilizing content similarity to improve I/O performance. In *Proceedings of the Eight USENIX Conference on File and Storage Technologies (FAST)*, pages 211–224, Feb. 2010.

[20] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller. Measurement and analysis of large-scale network file system workloads. In *Proceedings of the 2008 USENIX Annual Technical Conference*, pages 213–226, June 2008.

[21] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezis, and P. Camble. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FAST)*, pages 111–123, Feb. 2009.

[22] LSI. LSI WarpDrive SLP-300. `http://www.lsi.com/products/storagecomponents/Pages/WarpDriveSLP-300.aspx`, 2011.

[23] U. Manber. Finding similar files in a large file system. In *Proceedings of the Winter 1994 USENIX Technical Conference*, pages 1–10, Jan. 1994.

[24] T. McClure and B. Garrett. EMC Centera: Optimizing archive efficiency. Technical report, Jan. 2009.

[25] D. Meister and A. Brinkmann. dedupv1: Improving deduplication throughput using solid state drives (ssd). In *Proceedings of the 26th IEEE Symposium on Massive Storage Systems and Technologies*, pages 1–6, June 2010.

[26] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. In *Proceedings of the Ninth USENIX Conference on File and Storage Technologies (FAST)*, pages 1–13, Feb. 2011.

[27] Network Appliance Inc. Introduction to Data ONTAP 7G. Technical Report TR 3356, Network Appliance Inc.

[28] NIST. Secure hash standard SHS. Federal Information Processing Standards Publication FIPS PUB 180-3, Oct. 2008.

[29] Ocarina. Ocarina networks. `http://www.ocarinanetworks.com/`, 2011.

[30] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3: Design and implementation. In *Proceedings of the Summer 1994 USENIX Technical Conference*, pages 137–151, 1994.

[31] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, pages 89–101, Monterey, California, USA, 2002. USENIX.

[32] S. C. Rhea, R. Cox, and A. Pesterev. Fast, inexpensive content-addressed storage in Foundation. In *Proceedings of the 2008 USENIX Annual Technical Conference*, pages 143–156, June 2008.

[33] R. Rivest. The MD5 message-digest algorithm. Request For Comments (RFC) 1321, IETF, Apr. 1992.

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

[35] J. Satran. Internet small computer systems interface (iSCSI). Request For Comments (RFC) 3720, IETF, Apr. 2004.

[36] S. Silverberg. `http://opendedup.org`, 2011.

[37] Storage Networking Industry Association. Common Internet File System (CIFS) Technical Reference, 2002.

[38] StorWize. Preserving data integrity assurance while providing high levels of compression for primary storage. White paper, Mar. 2007.

[39] W. Xia, H. Jiang, D. Feng, and Y. Hua. Silo: a similarity-locality based near-exact deduplication scheme with low ram overhead and high throughput. In *Proceedings of the 2011 USENIX Annual Technical Conference*, pages 26–28, June 2011.

[40] L. L. You, K. T. Pollack, and D. D. E. Long. Deep Store: An archival storage system architecture. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*, Tokyo, Japan, Apr. 2005. IEEE.

[41] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the Sixth USENIX Conference on File and Storage Technologies (FAST)*, pages 269–282, Feb. 2008.

Caching less for better performance: Balancing cache size and update cost of flash memory cache in hybrid storage systems

Yongseok Oh[1], Jongmoo Choi[2], Donghee Lee[1], and Sam H. Noh[3]

[1]*University of Seoul, Seoul, Korea, {ysoh, dhl_express}@uos.ac.kr*
[2]*Dankook University, Gyeonggi-do, Korea, choijm@dankook.ac.kr*
[3]*Hongik University, Seoul, Korea, http://next.hongik.ac.kr*

Abstract

Hybrid storage solutions use NAND flash memory based Solid State Drives (SSDs) as non-volatile cache and traditional Hard Disk Drives (HDDs) as lower level storage. Unlike a typical cache, internally, the flash memory cache is divided into cache space and over-provisioned space, used for garbage collection. We show that balancing the two spaces appropriately helps improve the performance of hybrid storage systems. We show that contrary to expectations, the cache need not be filled with data to the fullest, but may be better served by reserving space for garbage collection. For this balancing act, we present a dynamic scheme that further divides the cache space into read and write caches and manages the three spaces according to the workload characteristics for optimal performance. Experimental results show that our dynamic scheme improves performance of hybrid storage solutions up to the off-line optimal performance of a fixed partitioning scheme. Furthermore, as our scheme makes efficient use of the flash memory cache, it reduces the number of erase operations thereby extending the lifetime of SSDs.

1 Introduction

Conventional Hard Disk Drives (HDDs) and state-of-the-art Solid State Drives (SSDs) each has strengths and limitations in terms of latency, cost, and lifetime. To alleviate limitations and combine their advantages, hybrid storage solutions that combine HDDs and SSDs are now available for purchase. For example, a hybrid disk that comprises the conventional magnetic disk with NAND flash memory cache is commercially available [30]. We consider hybrid storage that uses NAND flash memory based SSDs as a non-volatile cache and traditional HDDs as lower level storage. Specifically, we tackle the issue of managing the flash memory cache in hybrid storage.

The ultimate goal of hybrid storage solutions is providing SSD-like performance at HDD-like price, and achieving this goal requires near-optimal management of the flash memory cache. Unlike a typical cache, the flash memory cache is unique in that SSDs require *over-provisioned space (OPS)* in addition to the space for normal data. To make a clear distinction between OPS and space for normal data, we refer to the space in flash memory cache used to keep normal data as the *caching space*.

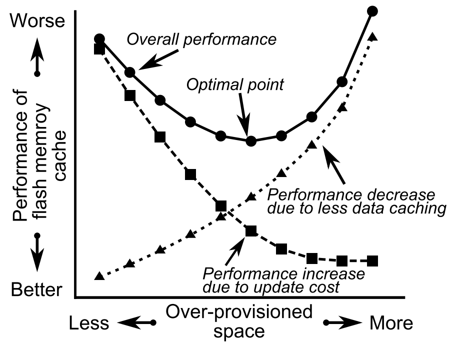


Figure 1: Balancing data in cache and update cost for optimal performance

The OPS is used for garbage collection operations performed during data updates. It is well accepted that given a fixed capacity SSD, increasing the OPS size brings about two consequences [11, 15, 26]. First, it reduces the caching space resulting in a smaller data cache. Less data caching results in decreased overall flash memory cache performance. Note Figure 1 (not to scale) where the x-axis represents the OPS size and the y-axis represents the performance of the flash memory cache. The dotted line with triangle marks shows that as the OPS size increases, caching space decreases and performance degrades.

In contrast, with a larger OPS, the update cost of data in the cache decreases and, consequently, performance of the flash memory cache improves. This is represented as the square marked dotted line in Figure 1. Note that as the two dotted lines cross, there exists a point where performance of the flash memory cache is optimal. The goal of this paper is to find this optimal point and use it in managing the flash memory cache.

To reiterate, the main contribution of this paper is in presenting a dynamic scheme that finds the workload dependent optimal OPS size of a given flash memory cache such that the performance of the hybrid storage system is optimized. Specifically, we propose cost models that are used to determine the optimal caching space and OPS sizes for a given workload. In our solution, the caching space is further divided into read and write caches, and we use cost models to dynamically adjust the sizes of the three spaces, that is, the read cache, write cache, and the OPS according to the workload for optimal hybrid storage performance. These cost models form the basis of the Optimal Partitioning Flash Cache Layer (OP-FCL) flash memory cache management scheme that we propose.

Experiments performed on a DiskSim-based hybrid storage system using various realistic server workloads show that OP-FCL performs comparatively to the offline optimal fixed partitioning scheme. The results indicate that caching as much data as possible is not the best solution, but caching an appropriate amount to balance the cache hit rate and the garbage collection cost is most appropriate. That is, caching less data in the flash memory cache can bring about better performance as the gains from reduced overhead for data update compensates for losses from keeping less data in cache. Furthermore, our results indicate that as our scheme makes efficient use of the flash memory cache, OP-FCL can significantly reduce the number of erase operations in flash memory. For our experiments, this results in the lifetime of SSDs being extended by as much as three times compared to conventional uses of SSDs.

The rest of the paper is organized as follows. In the next section, we discuss previous studies that are relevant to our work with an emphasis on the design of hybrid storage systems. In Section 3, we start off with a brief review of the HDD cost model. Then, we move on and describe cost models for NAND flash memory storage. Then, in Section 4, we derive cost models for hybrid storage and discuss the existence of optimal caching space and OPS division. We explain the implementation issues in Section 5 and then, present the experimental results in Section 6. Finally, we conclude with a summary and directions for future work.

2 Related Work

Numerous hybrid storage solutions that integrate HDDs and SSDs have been suggested [8, 11, 14, 29]. Kgil et al. propose splitting the flash memory cache into separate read and write regions taking into consideration the fact that read and write costs are different in flash memory [11]. Chen et al. propose Hystor that integrates low-cost HDDs and high-speed SSDs [4]. To make better use of SSDs, Hystor identifies critical data, such as metadata, keeping them in SSDs. Also, it uses SSDs as a write-back buffer to achieve better write performance. Pritchett and Thottethodi observe that reference patterns are highly skewed and propose a highly-selective caching scheme for SSD cache [26]. These studies try to reduce expensive data allocation and write operations in flash memory storage as writes are much more expensive than reads. They are similar to ours in that flash memory storage is being used as a cache in hybrid storage solutions and that some of them split the flash memory cache into separate regions. However, our work is unique in that it takes into account the trade-off between caching benefit and data update cost as determined by the OPS size.

The use of the flash memory cache with other objectives in mind have been suggested. As SSDs have lower energy consumption than HDDs, Lee et al. propose an SSD-based cache to save energy of RAID systems [18]. In this study, an SSD is used to keep recently referenced data as well as for write buffering. Similarly, to save energy, Chen et al. suggest a flash memory based cache for caching and prefetching data of HDDs [3]. Saxena et al. use flash memory as a paging device for the virtual memory subsystem [28] and Debnath et al. use it as a metadata store for their de-duplication system [5]. Combining SSDs and HDDs in the opposite direction has also been proposed. A serious concern of flash memory storage is its relatively short lifetime and, to extend SSD lifetime, Soundararajan et al. suggest a hybrid storage system called Griffin, which uses HDDs as a write cache [32]. Specifically, they use a log-structured HDD cache, periodically destaging data to SSDs so as to reduce write requests and, consequently, to increase the lifetime of SSDs.

There have been studies that concentrate on finding cost-effective ways to employ SSDs in systems. To satisfy high-performance requirements at a reasonable cost budget, Narayanan et al. look into whether replacing disk based storage with SSDs may be cost effective; they conclude that replacing disks with SSDs is not yet so [22]. Kim et al. suggest a hybrid system called HybridStore that combines both SSDs and HDDs [15]. The goal of this study is in finding the most cost-effective configuration of SSDs and HDDs.

Besides studies on flash memory caches, there are many buffer cache management schemes that use the idea of splitting caching space. Kim et al. present a buffer management scheme called Unified Buffer Management (UBM) that detects sequential and looping references and stores those blocks in separate regions in the buffer cache [13]. Park et al. propose CRAW-C (Clock for Read And Write considering Compressed file system) that allocates three memory areas for read, write, and compressed pages, respectively [24]. Shim et al. suggest an adaptive partitioning scheme for the DRAM buffer in SSDs. This scheme divides the DRAM buffer into the caching and mapping spaces, dynamically adjusting their sizes according to the workload characteristics [31]. This study is different from ours in that the notion of OPS is necessary for flash memory updates, while for DRAM, it is not.

3 Flash Memory Cache Cost Model

In this section, we present the cost models for SSDs and HDDs [35]. HDD reading and writing are characterized by seek time and rotational delay. Assume that C_{D_RPOS} and C_{D_WPOS} are sums of the average seek time and the average rotational delay for HDD reads and writes, respectively. Let us also assume that P is the data size in bytes and B is the bandwidth of the disk. Then, the data read and write cost of a HDD is derived as $C_{DR} = C_{D\_RPOS} + \frac{P}{B}$ and $C_{DW} = C_{D\_WPOS} + \frac{P}{B}$, respectively. (Detailed derivations are referred to Wang [35].)

Before moving on to the cost model of flash memory based SSDs, we give a short review of NAND flash memory and the workings of SSDs. NAND flash memory, which is the storage medium of SSDs, consists of a number of blocks and each block consists of a number of pages. Reads are done in page units and take constant time. Writes are also done in page units, but data can be written to a page only after the block containing the page becomes clean, that is, after it is erased. This is called the erase-before-write property. Due to this property, data update is usually done by relocating new data to a clean page of an already erased block and most flash memory storage devices employ a sophisticated software layer called the Flash Translation Layer (FTL) that relocates modified data to new locations. The FTL also provides the same HDD interface to SSD users. Various FTLs such as page mapping FTL [7, 34], block mapping FTL [12], and many hybrid mapping FTLs [10, 17, 19, 23] have been proposed. Among them, the page mapping FTL is used in many high-end commercial SSDs that are used in hybrid storage solutions. Hence, in this paper, we focus on the page mapping FTL. However, the methodology that follows may be used with block and hybrid mapping FTLs as well. The key difference would be in deriving garbage collection and page write cost models appropriate for these FTLs.

As previously mentioned, the FTL relocates modified data to a clean page, and pages with old data become invalid. The FTL recycles blocks with invalid pages by performing garbage collection (GC) operations. For data updates and subsequent GCs, the FTL must always preserve some number of empty blocks. As data updates consume empty blocks, the FTL must produce more empty blocks by performing GCs that collect valid pages scattered in used blocks to an empty block, marking the used blocks as new empty blocks. The worst case and average GC costs are determined by the ratio of the initial OPS to the total storage space. It has been shown that the worst case and average GC costs become lower as more over-provisioned blocks are reserved [9].

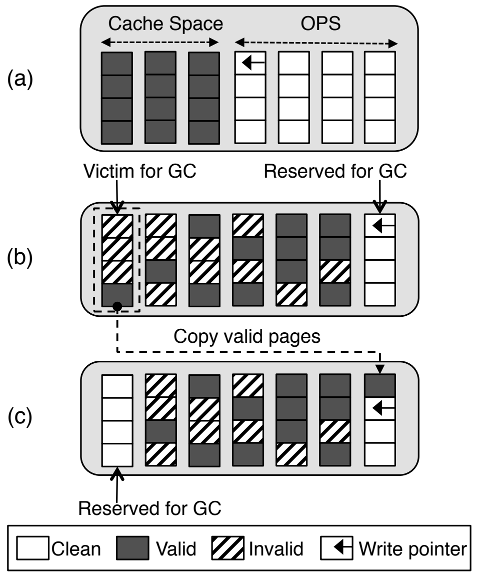


Figure 2: Garbage collection in flash memory storage

If we assume that the FTL selects the block with the minimum number of valid pages for a GC operation, then the worst case GC occurs when all valid (or invalid) pages are evenly distributed to all flash memory blocks except for an empty block that is preserved for GC operations. For now, let us assume that u is the worst case utilization determined from the initial number of over-provisioned blocks and data blocks. Then, in Fig. 2(a), where there are 3 data blocks containing cached data and 4 initial over-provisioned blocks, the worst case u is calculated as $3/(3+4-1)$. (We subtract 1 because the FTL must preserve one empty block for GC as marked by the arrow in Fig. 2(b).) From u, the maximum number of valid pages in the block selected for GC can be derived as $\lceil u \cdot N_P \rceil$, where N_P is the number of pages in a block.

Then, the worst case GC cost for a given utilization u can be calculated from the following equation, where N_P is the number of pages in a block, C_E is the erase cost (time) of a flash memory block, and C_{CP} is the page copy cost (time). (We assume that the copyback operation is being used. For flash memory chips that do not support copyback, C_{CP} may be expanded to a sequence of read, C_{PR} , and write, C_{PROG}, operations.)

$$C_{GC}(u) = \lceil u \cdot N_P \rceil \cdot C_{CP} + C_E \qquad (1)$$

That is, as seen in Fig. 2(b) and (c), a GC operation erases an empty block with cost C_E and copies all valid pages from the block selected for GC to the erased empty block with cost $\lceil u \cdot N_P \rceil \cdot C_{CP}$. Then, the garbage-collected block becomes an empty block that may be used for the next GC. The remaining clean pages in the previously empty block are used for subsequent write requests. If all those clean pages are consumed, then another GC operation will be performed.

After GC, in the worst case, there are $\lfloor (1-u) \cdot N_P \rfloor$ clean pages in what was previously an empty block (for example, the right-most block in Fig. 2(c)) and write requests of that number can be served in the block. Let us assume that C_{PROG} is the page program time (cost) of flash memory. (Note that "page program" and "page write" are used interchangeably in the paper.) By dividing GC cost and adding it to each write request, we can derive, $C_{PW}(u)$, the page write cost for worst case utilization u as follows.

$$C_{PW}(u) = \frac{C_{GC}(u)}{\lfloor (1-u) \cdot N_P \rfloor} + C_{PROG} \qquad (2)$$

Equation 2 is the worst case page update cost of flash memory storage assuming valid data (or invalid data) are evenly distributed among all the blocks. Typically, however, the number of valid pages in a block will vary. For example, the block marked "Victim for GC" in Fig. 2(b) has a smaller number of valid pages than the other blocks. Therefore, in cases where the FTL selects a block with a small number of valid pages for the GC operation, then utilization of the garbage-collected block, u', would be lower than the worst case utilization, u. Previous LFS and flash memory studies derived and used the following relation between u' and u [17, 20, 35].

$$u = \frac{u'-1}{ln\, u'}$$

Let $U(u)$ be the function that translates u to u'. (In our implementation, we use a table that translates u to u'.) Then the average page update cost can be derived by applying $U(u)$ for u in Equation 1 and 2 leading to Equation 3 and 4.

$$C_{GC}(u) = U(u) \cdot N_P \cdot C_{CP} + C_E \qquad (3)$$

$$C_{PW}(u) = \frac{C_{GC}(u)}{(1-U(u)) \cdot N_P} + C_{PROG} \qquad (4)$$

4 Hybrid Storage Cost Model

In the previous section, the garbage collection and page update cost of flash memory storage was derived. In this section, we derive the cost models for hybrid storage systems, which consist of a flash memory cache and a HDD. Specifically, the cost models determine the optimal size of the caching space and OPS minimizing the overall data access cost of the hybrid storage system. In our derivation of the cost models, we first derive the read cache cost model and then, derive the read/write cache cost model used to determine the read cache size, write cache size and OPS size. Our models assume that the cache management layer can measure the hit and miss rates of read/write caches as well as the number of I/O requests. These values can be easily measured in real environments.

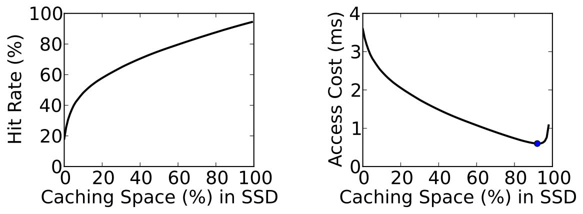
4.1 Read cache cost model

On a read request the storage examines whether the requested data is in the flash memory cache. If it is, the storage reads it and transfers it to the host system. If it is not in the cache, the system reads it from the HDD, stores it in the flash memory cache and transfers it to the host system. If the flash memory cache is already full with data (as will be the case in steady state), it must invalidate the least valuable data in the cache to make room for the new data. We use the LRU (Least Recently Used) replacement policy to select the least valuable data. In the case of read caching, the selected data need only be invalidated, which can be done essentially for free. (We discuss the issue of accommodating other replacement policies in Section 5.)

Let us assume that $H_R(u)$ is the cache read hit rate for a given cache size, which is determined by the worst case utilization u, as we will see later. With rate $H_R(u)$, the system reads the requested data from the cache with cost C_{PR}, the page read operation cost (time) of flash memory, and transfers it to the host system. With rate $1-H_R(u)$, the system reads data from disk with cost C_{DR} and, after invalidating the least valuable data selected by the cache replacement policy, stores it in the flash memory cache with cost $C_{PW}(u)$, which is the cost of writing new data to cache including the possible garbage collection cost. Then, C_{HR}, the read cost of the hybrid storage system with a read cache, is as follows.

$$C_{HR}(u) = H_R(u) \cdot C_{PR} + (1-H_R(u)) \cdot (C_{DR} + C_{PW}(u)) \qquad (5)$$

Let us now take the flash memory cache size into consideration. For a given flash memory cache size, S_F, the read cache size, S_R and the OPS size S_{OPS} can be



(a) Read hit rate (b) Read cost

Figure 3: (a) Read hit rate curve generated using the numpy.random.zipf Python function (Zipfian distribution with $\alpha = 1.2$ and range = 120%) and (b) the hybrid storage read cost graph for this particular hit rate curve, with optimal point at 92%.

approximated from u such that $S_{OPS} \approx (1-u) \cdot S_F$ and $S_R \approx u \cdot S_F$. These sizes are approximated values as they do not take into account the empty block reserved for GC. (Recall the empty block in Fig. 2.) Though calculating the exact size is possible by considering the empty block, we choose to use these approximations as these are simpler, and their influence is negligible relative to the overall performance estimation.

Let us now take an example. Assume that we have a hit rate curve $H_R(u)$ for read requests as shown in Fig. 3(a), where the x-axis is the cache size and the y-axis is the hit rate. Then, with Equation 5, we can redraw the hit rate curve with u on the x-axis, and consequently, the access cost graph of the hybrid storage system becomes Fig. 3(b). The graph shows that the overall access cost becomes lower as u increases until u reaches 92%, where the access cost becomes minimal. Beyond this point, the access cost suddenly increases, because even though the caching benefit is still high the data update cost soars as the OPS shrinks. Once we find u with minimum cost, the read cache size and OPS size can be found from $S_{OPS} \approx (1-u) \cdot S_F$ and $S_R \approx u \cdot S_F$.

4.2 Read and write cache cost model

Previous studies have shown that due to their difference in costs, separating read and write requests in flash memory storage has a significant effect on performance [11]. Hence, we now incorporate write cost to the model by dividing the flash caching space into two areas, namely a write cache and a read cache. The read cache, whose cost model was derived in the previous subsection, contains data that has recently been read but never written back while the write cache keeps data that has recently been written, but not yet destaged. Therefore, data in the write cache are dirty and they must be written to the HDD when evicted from the cache. When a write is requested to data in the read cache, we regard it as a write miss. In this case, we invalidate the data in the read cache and write the new data in the write cache. We consider the case of reading data in the write cache later.

In the following cost model derivation, we assume write-back policy for the write cache. This choice is more efficient than the write-through policy without any loss in consistency as the flash cache is also non-volatile. If the write-through policy must be used, our model needs to be modified to reflect the additional write to HDD that is incurred for each write to the flash cache. This will result in a far less efficient hybrid storage system.

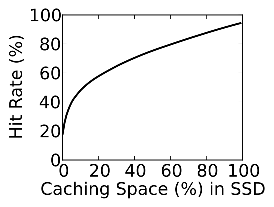
There can be three types of requests to the flash write cache. The first is a write hit request, which is a write request to existing data in the write cache. In this case, the old data becomes invalidated and the new data is written to the write cache with cost $C_{PW}(u)$. The second is a write miss request, which is a write request to data that does not exist in the write cache. In this case, the cache replacement policy selects victim data that should be read from the write cache and destaged to the HDD with cost $C_{PR} + C_{DW}$ to make room for the newly request data. (Note we are assuming the system is in steady state.) After evicting the data, the hybrid storage system writes the new data to the write cache with cost $C_{PW}(u)$. The last type of request is a read hit request, which is a read request to existing (and possibly dirty) data in the write cache. This happens when a read request is to data that is already in the write cache. In this case, the request can be satisfied with cost C_{PR}, that is, the flash memory page read cost. Note that there is no read miss request to the write cache because read requests to data not in cache are handled by the read cache.

Now we introduce a parameter r, which is the read cache size ratio within the caching space, where $0 \leq r \leq 1$. Naturally, $1-r$ is the ratio of the write cache size. If r is 1, all caching space is used as a read cache and, if it is 0, all caching space is used as a write cache. Let S_C denote the total size of the caching space. Then, we can calculate the read cache size, S_R, and write cache size, S_W, from S_C such that $S_R = S_C \cdot r$ and $S_W = S_C \cdot (1-r)$. Note that S_C is calculated from u such that $S_C \approx u \cdot S_F$. Then, S_R and S_W are determined by u and r.

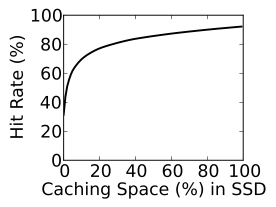
Let us assume that the cache management layer can measure the read hit rates of the read cache and draw $H_R(u,r)$, the read cache hit rate curve, which now has two parameters u and r. (We will show that the hit rate curve can be obtained by using ghost buffers in the next section.) Then, the read cost of the hybrid storage system is now modified as follows.

$$\begin{aligned} C_{HR}(u,r) = (1-H_R(u,r)) \cdot (C_{DR} + C_{PW}(u)) \\ +H_R(u,r) \cdot C_{PR} \end{aligned}$$

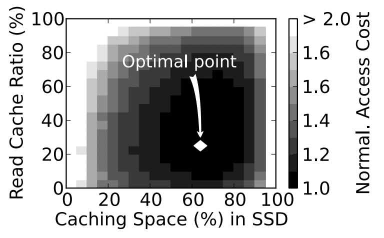
Let us also assume that we can measure the write hit, the write miss, and the read hit rates of the write cache



(a) Read hit rate



(b) Write hit rate



(c) Expected access cost

Figure 4: (a) Read and (b) write hit rate curves generated using the numpy.random.zipf Python function ((a) Zipfian distribution with $\alpha = 1.2$ and range = 120%, (b) Zipfian distribution with $\alpha = 1.4$ and range = 220%) and (c) the hybrid storage access cost graph for these hit rate curves.

and draw the hit rate curves. For the moment, let us regard the read hit in the write cache as being part of the write hit. Assume that $H_W(u,r)$ is the write cache hit rate for a given write cache size, and it has two parameters that determine the cache size. Then, with rate $H_W(u,r)$, a write request finds its data in the write cache, and the cost of this action is $H_W(u,r) \cdot C_{PW}(u)$. Otherwise, with rate of $1 - H_W(u,r)$, the write request does not find data in the write cache. Servicing this request requires reading and evicting existing data and writing new data to the write cache. Hence, the cost is $(1 - H_W(u,r)) \cdot (C_{PR} + C_{DW} + C_{PW}(u))$. In summary, the write cost of the hybrid storage system can be given as follows.

$$\begin{aligned} C_{HW}(u,r) = (1 - H_W(u,r)) \\ \cdot (C_{PR} + C_{DW} + C_{PW}(u)) \\ + H_W(u,r) \cdot C_{PW}(u) \end{aligned}$$

Now let us consider the read hit case within the write cache. Although it is possible to maintain separate read hit and write hit curves for the write cache, this makes the cost model more complex without much benefits, especially in terms of implementation. Therefore, we devise a simple approximation method for incorporating the read hit case in the write cache. Assume that h' is the read hit rate in the write cache. (Then, naturally, $1 - h'$ is the write hit rate in the write cache.) Then, with rate h', the read hit is satisfied with cost C_{PR} and with rate $1 - h'$, the write hit is satisfied with cost $C_{PW}(u)$. Now we can calculate the average cost for both read hit and write hit such that $C_{WH} = (1 - h') \cdot C_{PW}(u) + h' \cdot C_{PR}$. By assuming $H_W(u,r)$ is the hit rate including both read and write hits, the write cost of the hybrid storage system now can be given as follows.

$$\begin{aligned} C_{HW}(u,r) = (1 - H_W(u,r)) \\ \cdot (C_{PR} + C_{DW} + C_{PW}(u)) \\ + H_W(u,r) \cdot C_{WH} \end{aligned}$$

Now, let IO_R and IO_W, respectively, be the rate served in the read and write caches among all requests. For example, of a total of 100 requests, if 70 requests are served in the read cache and 30 requests are served in the write cache, then IO_R is 0.7 and IO_W is 0.3. Then we can derive, $C_{HY}(u,r)$, the overall access cost of the hybrid storage system that has separate read and write caches and OPS as follows.

$$\begin{aligned} C_{HY}(u,r) = C_{HR}(u,r) \cdot IO_R + \\ C_{HW}(u,r) \cdot IO_W \end{aligned} \tag{6}$$

Let us take an example. Assume that, at a certain time, the hybrid storage system finds IO_R, IO_W, h' to be 0.2, 0.8, and 0.2, respectively, and the read and write hit rate curves are estimated as shown in Fig. 4(a) and (b). In the graph, both read and write hit rates increase as caches become larger but slowly saturate beyond some point. As the read and write cache sizes are determined by u and r, we can obtain the read and write cache hit rates for given u and r values from the hit rate curves. Then, with the cost model of Equation 6, we can draw the overall access cost graph of the system as in Fig. 4(c). In the graph, the x-axis is u and the y-axis is r. These two parameters determine the read and write cache sizes as well as the OPS size. In Fig. 4(c), darker shades reflect lower access cost and we pinpoint the lowest access cost with the diamond mark pointed to by the arrow.

Specifically, the minimum overall access cost of the hybrid storage system is when u is 0.64 and r is 0.25 for this particular configuration. For a 4GB flash memory cache, this translates to the read cache size of 0.64GB, the write cache size of 1.92GB, and an OPS size of 1.44GB.

5 Implementation Issues of Flash Cache Layer

In this section, we describe some implementation issues related to our flash memory cache management scheme, which we refer to as OP-FCL (Optimal Partitioning of Flash Cache Layer). Fig. 5(a) shows the overall structure of the hybrid storage system that we envision. The storage system has a HDD serving as main storage and an SSD, which we also refer to as the flash cache layer (FCL), that is used as a non-volatile cache keeping recently read/written data as previous studies have done [4, 11, 15]. As is common on SSDs, it has DRAM for buffering I/O data and storing data structures used by the SSD. The space at the flash cache layer is divided into three regions: the read cache area, the write cache area, and the over-provisioned space (OPS) as shown in Fig. 5(b). OP-FCL measures the read and write cache hit and miss rates and the I/O rates. Then, it periodically calculates the optimal size of these cache spaces and progressively adjusts their sizes during the next period.

To accurately simulate the operations and measure the costs of the hybrid storage system, we use DiskSim [2] to emulate the HDD and DiskSim's MSR SSD extension [1] to emulate the SSD. Specifically, the simulator mimics the behaviour of Maxtor's Atlas 10K IV disk whose average read and write latency is $4.4ms$ and $4.9ms$, respectively, with transfer speed of $72MB/s$. Also, the SSD simulator emulates SLC NAND flash memory chip operations, and it takes $25us$ to read a page, $200us$ to write a page, $1.5ms$ to erase a block, and $100us$ to transfer data to/from a page of flash memory through the bus. The page and block unit size is 4KB and 256KB, respectively, and the flash cache layer manages data in 4KB units.

In the simulator, we modified the SSD management modules and implemented additional features as well as the OP-FCL. OP-FCL consists of several components, namely, the *Page Replacer*, *Sequential I/O Detector*, *Workload Tracker*, *Partition Resizer*, and *Mapping Manager*.

The Page Replacer has two LRU lists, one each for the read and write caches, and maintains LRU ordering of data in the caches. In steady state when the cache is full, the LRU data is evicted from the cache to accommodate newly arriving data. For the read cache, cache eviction simply means that the data is invalidated, while for write cache, it means that data must be destaged, incurring a flash cache layer read and a disk write operation. In the actual implementation, the Page Replacer destages several dirty data altogether to minimize seek distance by applying the elevator disk scheduling algorithm. However, we do not consider group destaging in our cost model as it has only minimal overall impact. This is because the number of data destaged as a group is relatively small compared to the total number of data in the write cache.

Previous studies have taken notice of the impact of sequential references on cache pollution and thus, have tried to detect and treat them separately [13]. The Sequential I/O Detector monitors the reference pattern and detects sequential references. In our current implementation, consecutive I/O requests greater than $128KB$ are regarded as sequential references, and those requests bypass the flash cache layer and are sent directly to disk to avoid cache pollution.

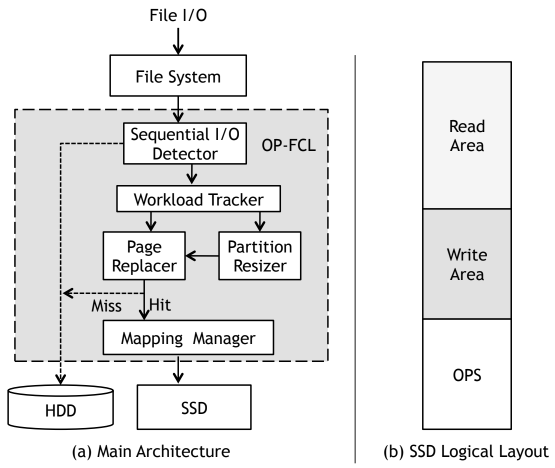


Figure 5: OP-FCL architecture

Besides the Page Replacer that manages the cached data, the Workload Tracker maintains LRU lists of ghost buffers to simultaneously measure hit rates of various cache sizes, following the method proposed by Patterson et al. [25]. Ghost buffers maintain only logical addresses, not the actual data and, thus, memory overhead is minimal requiring roughly 1% of the total flash memory cache. Part of the ghost buffer represents data in cache and others represent data that have already been evicted from the cache. Keeping information of evicted data in the ghost buffer makes it possible to measure the hit rate of a cache larger than the actual cache size. To simulate various cache sizes simultaneously, we use N-segmented ghost buffers. In other words, we divide the ghost buffer into N-segments corresponding to N cache sizes and thus, hit rates of N cache sizes can be obtained by combining the hit rates of the segments. From the hit rates of N cache sizes, we obtain the read/write hit rate curves by interpolating the missing cache sizes.

Note that though we use the LRU cache replacement policy for this study, our model can accommodate any replacement policy so long as they can be implemented in the flash cache and the ghost buffer management layers. Different replacement policies will generate different read/write hit rate curves and, in the end, affect the results. However, a replacement policy only affects the read/write hit rate curves, and thus, our overall cost model is not affected.

These hit rate curves are obtained per period. In the current implementation, a period is the logical time to process 65536 (2^{16}) read and write requests. When the period ends, new hit rate curves are generated, while a

Algorithm 1 Optimal Partitioning Algorithm

```
1:  procedure OPTIMAL_PARTITIONING
2:      $step \leftarrow segment\_size/total\_cache\_size$
3:      INIT_PARMS($op\_cost$, $op\_u$, $op\_r$)
4:      for $u \leftarrow step$; $u < 1.0$; $u \leftarrow u + step$ do
5:          for $r \leftarrow 0.0$; $r \leq 1.0$; $r \leftarrow r + step$ do
6:              $cur\_cost \leftarrow C_{HY}(u, r)$                ▷ Call Eq. 6
7:              if $cur\_cost < op\_cost$ then
8:                  $op\_cost \leftarrow cur\_cost$
9:                  $op\_u \leftarrow u$, $op\_r \leftarrow r$
10:             end if
11:         end for
12:     end for
13:     ADJUST_CACHE_SIZE($op\_u$, $op\_r$)
14: end procedure
```

new period starts. Then, with the hit rate curves generated by the Workload Tracker in the previous period, the Partition Resizer gradually adjusts the sizes of the three spaces, that is, the read and write cache space and the OPS for the next period. To make the adjustment, the Partition Resizer determines the optimal u and r as described in Section 4, and those optimal values in turn decide the optimal size of the three spaces.

To obtain the optimal u and r, we devise an iterative algorithm presented in Algorithm 1. Starting from u=step, the outer loop iterates the inner loop increasing u in 'step' increments while u is less than 1.0. The two extreme configurations that we do not consider are where OPS is 0% and 100%. These are unrealistic configurations as OPS must be greater than 0% to perform garbage collection, while OPS being 100% would mean that there is no space to cache data. The inner loop starting from r=0 iterates, calculating the access cost of the hybrid storage system as derived in Equation 6, while increasing r in 'step' increments until r becomes greater or equal to 1.0. The 'step' value can be calculated as the segment size divided by the total cache size, as shown in the second line of Algorithm 1. The nested loop iterates $N \times M$ times to calculate the costs, where N is the outer loop count, 1/step-1, and M is the inner loop count, 1/step+1. A single cost calculation consists of 10 ADD, 4 SUB, 11 MUL, and 4 DIV operations. Finer 'step' values may be used resulting in finer u and r values, but with increased cost calculation overhead. However, computational overhead for executing this algorithm is quite small because they run once every period and the calculations are just simple arithmetic operations.

Once the optimal u and r and, in turn, the optimal sizes are determined, the Partition Resizer starts to progressively adjust the sizes of the three spaces. To increase OPS size, it gradually evicts data in the read or write caches. To increase cache space, that is, decrease OPS, GC is performed to produce empty blocks. These empty blocks are then used by the read and/or write caches.

The key role of our Mapping Manager is translating the logical address to a physical location in the flash cache layer. For this purpose, it maintains a mapping table that keeps the translation information. In our implementation, we keep the mapping information at the last page of each block. As we consider flash memory blocks with 64 pages, the overhead is roughly 1.6%. Moreover, we implement a crash recovery mechanism similar to that of LFS [27]. If a power failure occurs, it searches for the most up-to-date checkpoint and goes through a recovery procedure to return to the checkpoint state.

6 Performance Evaluation

In this section, we evaluate OP-FCL. For comparison, we also implement two other schemes. The first is the Fixed Partition-Flash Cache Layer (FP-FCL) scheme. This is the simplest scheme where the read and write cache is not distinguished, but unified as a single cache. The OPS is available with a fixed size. This scheme is used to mimic a typical SSD of today that may serve as a cache in a hybrid storage system. Normally, the SSD would not distinguish read and write spaces and it would have some OPS, whose size would be unknown. We evaluate this scheme as we vary the percentage of the caching space set aside for the (unified) cache. The best of these results will represent the most optimistic situation in real life deployment.

The other scheme is the Read and Write-Flash Cache Layer (RW-FCL) scheme. This scheme is in line with the observation made by Kgil et al. [11] in that read and write caches are distinguished. This scheme, however, goes a step further in that while the sum of the two cache sizes remain constant, the size between the two are dynamically adjusted for best performance according to the cost models described in Section 4. For this scheme, the OPS size would also be fixed as the total read and write cache size is fixed. We evaluate this scheme as we vary the percentage of the caching space set aside for the combined read and write cache. Initial, all three schemes start with an empty data cache. For OP-FCL, the initial OPS size is set to 5% of the total flash memory size.

The experiments are conducted using two sets of traces. We categorize them based on the size of requests. The first one, 'Small Scale', are workloads that request less than 100GBs of total data. The other set, 'Large Scale', are workloads with over 100GBs of data requests. Details of the characteristics of these workloads are in Table 1.

The first two subsections discuss the performance aspects of the two class of workloads. Then, in the next

Type	Workload	Working Set Size (GB)			Avg. Req. Size (KB)		Request Amount (GB)		Read Ratio
		Total	Read	Write	Read	Write	Read	Write	
Small Scale	Financial [33]	3.8	1.2	3.6	5.7	7.2	6.9	28.8	0.19
	Home [6]	17.2	13.5	5.0	22.2	3.9	15.3	66.8	0.18
	Search Engine [33]	5.4	5.4	0.1	15.1	8.6	15.6	0.001	0.99
Large Scale	Exchange [22]	79.35	74.12	23.29	9.89	12.4	114.36	131.69	0.46
	MSN [22]	37.98	30.93	23.03	11.48	11.12	107.23	74.01	0.59

Table 1: Characteristics of I/O workload traces

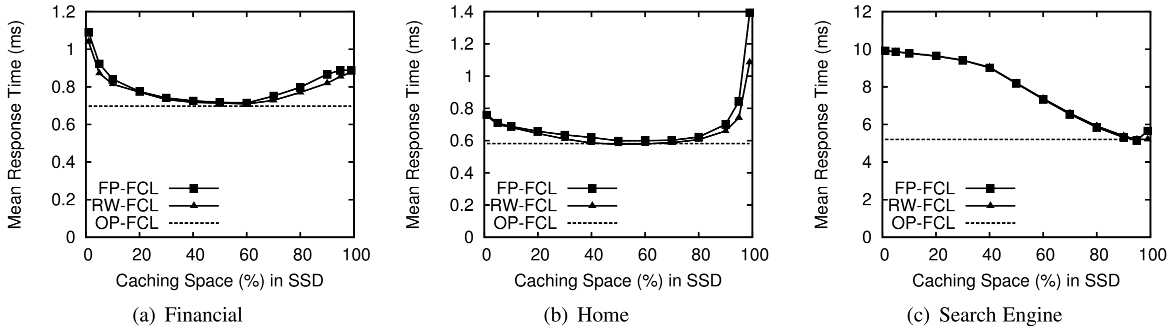


Figure 6: Mean response time of hybrid storage

Type	Description	Config. 1	Config. 2
OP-FCL	N_P	64	
	C_{PROG}	300us	
	C_{PR}	125us	
	C_{CP}	225us	
	C_E	1.5ms	
	C_{D_RPOS}	4.5ms	
	C_{D_WPOS}	4.9ms	
	B	72MB/s	
	P	4KB	
	segment_size	256MB	
SSD	Total Capacity	4GB	16GB
	No. of Packages	1	4
	Blocks Per Package	16384	
	Planes Per Package	1	
	Cleaning Policy	Greedy	
	GC Threshold	1%	
	Copyback	On	
HDD	Model	Maxtor Atlas 10K IV	
	No. of Disks	1	3

Table 2: Configuration of Hybrid Storage System

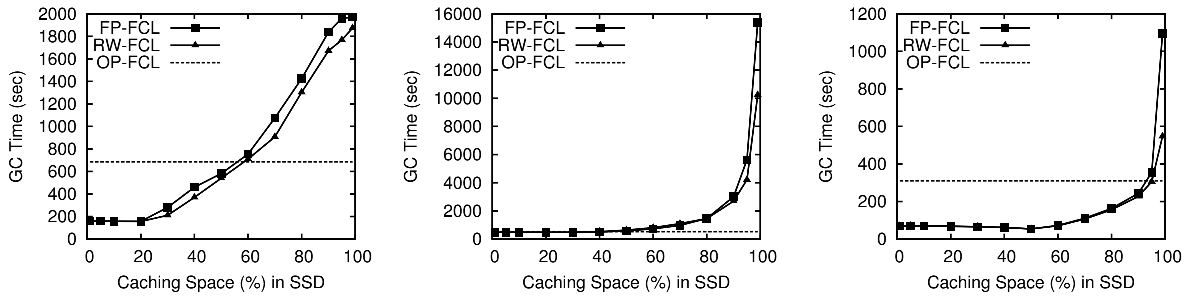
subsection, we present the effect of OP-FCL on the lifetime of SSDs. In the final subsection, we present a sensitivity analysis of two parameters that needs to be determined for our model.

6.1 Small scale workloads

The experimental setting is as given in Fig. 5 described in Section 5. The specific configuration of the HDD and SSD used in these experiments is shown in Table 2 denoted as 'Config. 1'. All other parameters not explicitly mentioned are set to default values. We assume a single SSD is employed as the flash memory cache and a single HDD as the main storage. This configuration is similar to that of a real hybrid drive [30].

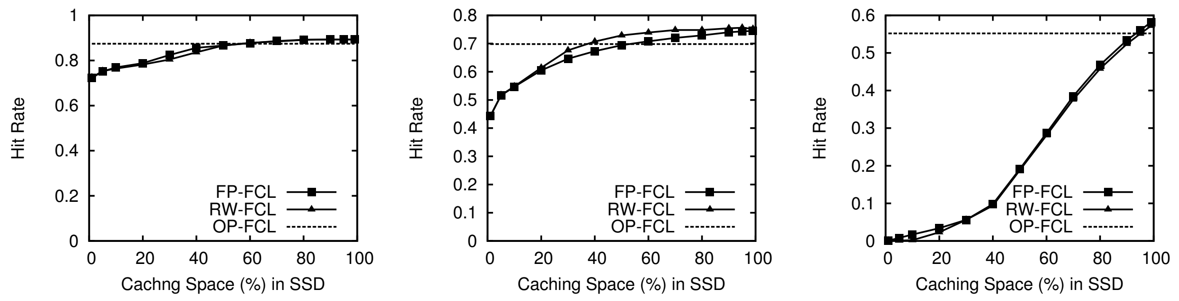
For small scale workloads we use three traces, namely, Financial, Home, and Search Engine that have been used in numerous previous studies [7, 11, 15, 16, 17]. The Financial trace is a random write intensive I/O workload obtained from an OLTP application running at a financial institutions [33]. The Home trace is a random write intensive I/O workload obtained from an NFS server that keeps home directories of researchers whose activities are development, testing, and plotting [6]. The Search Engine trace is a random read intensive I/O workload obtained from a web search engine [33]. The Search Engine trace is unique in that 99% of the requests are reads while only 1% are writes.

Fig. 6 shows the results of the cache partitioning schemes, where the measure is the response time of the hybrid storage system. The *x*-axis here denotes the ratio of caching space (unified or read and write combined) for the FP-FCL and RW-FCL schemes. For example, 60 in the *x*-axis means that 60% of the flash memory capacity is used as caching space and 40% is used as OPS. The *y*-axis denotes the average response time of the read and write requests. In the figure, the response times of FP-FCL and RW-FCL schemes vary according to the ratio of the caching space. In contrast, the response time of OP-FCL is drawn as a horizontal line because it reports



(a) Financial (b) Home (c) Search Engine

Figure 7: Cumulative garbage collection time



(a) Financial (b) Home (c) Search Engine

Figure 8: Hit rate

only one response time regardless of the ratio of caching space as it dynamically adjusts the three spaces according to the workload.

Let us first compare FP-FCL and RW-FCL in Fig. 6. In cases of the Financial and Home traces, we see that RW-FCL provides lower response time than FP-FCL. This is because RW-FCL is taking into account the different read and write costs in the flash memory cache layer. This result is in accord with previous studies that considered different read and write costs of flash memory [11]. However, in the case of the Search Engine trace, discriminating read and write requests has no effect because 99% of the requests are reads. Naturally, FP-FCL and RW-FCL show almost identical response times.

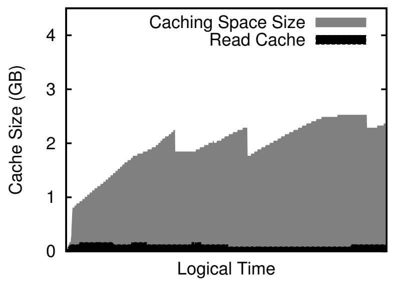
Now let us turn our focus to the relationship between the size of caching space (or OPS size) and the response time. In Fig. 6(a) and (b), we see that the response time decreases as the caching space increases (or OPS decreases) until it reaches the minimal point, and then increases beyond this point. Specifically, for FP-FCL and RW-FCL, the minimal point is at 60% for the Financial trace and at 50% for the Home trace for both schemes. In contrast, for the Search Engine trace, response time decreases continuously as the cache size increases and the optimal point is at 95%. The reason behind this is that the trace is dominated by read requests with rare modifications to the data. Thus, the optimal configuration for this trace is to keep as large a read cache as possible with only a small amount of OPS and write cache.

For the FP-FCL and RW-FCL schemes, the response time at the optimal point can be regarded as the off-line optimal value because it is obtained after exploring all possible configurations of the scheme. Let us now compare the response time of OP-FCL and the off-line optimal results of RW-FCL. In all traces, OP-FCL has almost the same response time as the off-line optimal value of RW-FCL. This shows that the cost model based dynamic adaptation technique of OP-FCL is efficient in determining the optimal OPS and the read and write cache sizes.

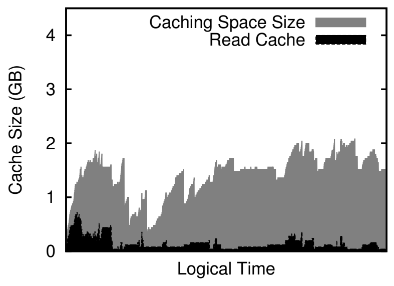
We now discuss the trade-off between garbage collection (GC) cost and the hit rate at the flash cache layer. Fig. 7 and 8 depict these results. In Fig. 7, we see that for all traces, GC cost increases, that is, performance degrades, continuously as caching space increases. The hit rate, on the other hand, increases, thus improving performance as caching space increases for all the traces as we can see in Fig. 8. For clear comparisons, we report the sum of the read and write hit rates for RW-FCL and OP-FCL. Note that both schemes measure read and write hit rates separately.

These results show the existence of two contradicting effects as caching space is increased, that is, increasing cache hit rate, which is a positive effect, and increasing GC cost, which is a negative effect. The interaction of these two contradicting effects leads to an optimal point where the overall access cost of the hybrid storage system becomes minimal.

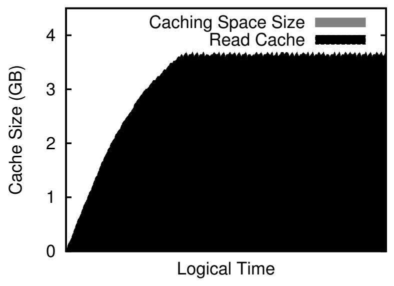
To investigate how well OP-FCL adjusts the caching



(a) Financial



(b) Home



(c) Search Engine

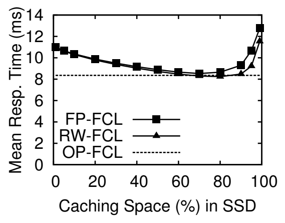
Figure 9: Dynamic size adjustment of read and write caches and OPS

space and OPS sizes, we continuously monitor their sizes as the experiments are conducted. Fig. 9 shows these results. In the figure, the *x*-axis denotes logical time that elapses upon each request and the *y*-axis denotes the total (read + write) caching space size and the read cache size. For the Financial and Home traces, we see that the caching space size increases and decreases repeatedly according to the reference pattern of each period as the cost models maneuver the caching space and OPS sizes. Notice that out of the 4GB of flash memory cache space, only 2 to 2.5GBs are being used for the Financial trace and less than half is used for the Home trace. Even though cache space is available, using less of it helps performance as keeping space to reduce garbage collection time is more beneficial. Note, though, that for the Search Engine trace, most of the 4GB are being allotted to the caching space, in particular, to the read cache. This is a natural consequence as reads are dominant, garbage collection rarely happens. Also note that it is taking some time for the system to stabilize to the optimal allocation setting.

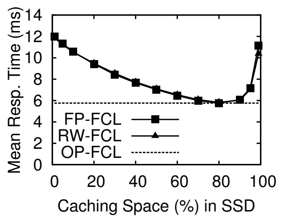
6.2 Large scale workloads

Our experimental setting for large scale workloads is as shown in Fig. 5 with the configuration summarized as 'Config. 2' in Table 2. In this configuration the SSD is 16GBs employing four packages of flash memory and the HDD consists of three 10K RPM drives.

To test our scheme for large scale enterprise workloads, we use the Exchange and MSN traces that have been used in previous studies [15, 21, 22]. The Exchange trace is a random I/O workload obtained from the Microsoft employee e-mail server [22]. This trace is composed of 9 volumes of which we select and use traces of volumes 2, 4, and 8, and each volume is assigned to each HDD. The MSN trace is extracted from 4 RAID-10 volumes on an MSN storage back-end file store [22]. We choose and use the traces of volumes 0, 1, and 4, each assigned to one HDD. The characteristics of the two traces are summarized in Table 1.

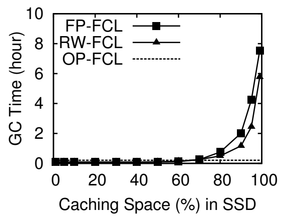


(a) Exchange

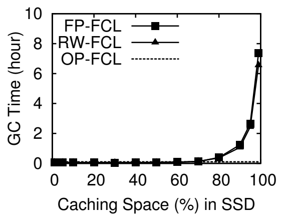


(b) MSN

Figure 10: Response time of hybrid storage

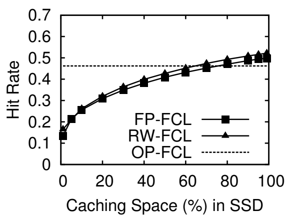


(a) Exchange

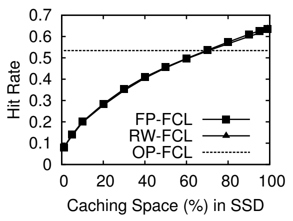


(b) MSN

Figure 11: Cumulative garbage collection time

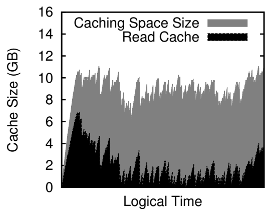


(a) Exchange

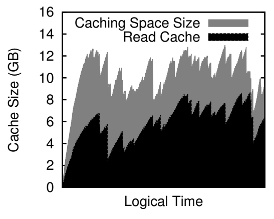


(b) MSN

Figure 12: Hit rate

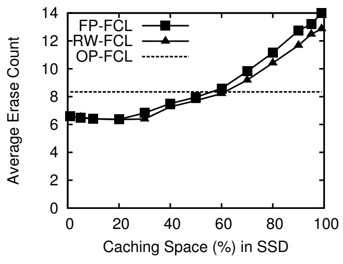


(a) Exchange

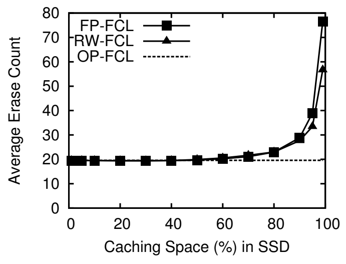


(b) MSN

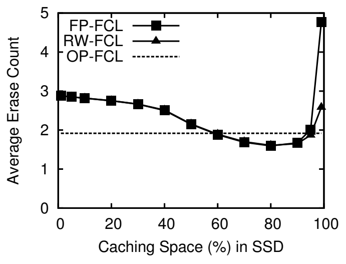
Figure 13: Dynamic size adjustment of read and write caches and OPS



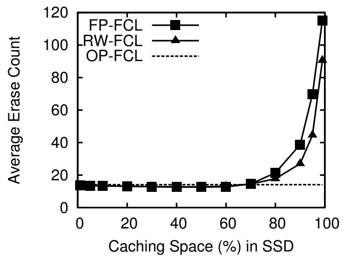
(a) Financial



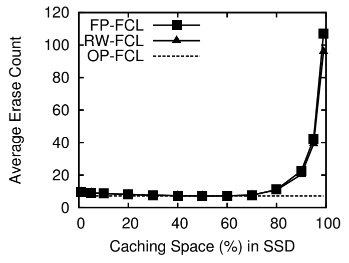
(b) Home



(c) Search Engine



(d) Exchange



(e) MSN

Figure 14: Average erase count of flash memory blocks

Fig. 10, which depicts the response time for the two large scale workloads, show similar trends that we observed with the small scale workloads, in that, as caching space increases, response time decreases to a minimal point, and then increases again. The response time of OP-FCL, which is shown as a horizontal line in the figure, is close to the smallest response times of FP-FCL and RW-FCL. From these results, we confirm again that a trade-off between GC cost and hit rate exists at the flash cache layer.

Specifically, for the Exchange trace shown in Fig. 10(a), the minimal point for FP-FCL is at 70%, while it is at 80% for RW-FCL. The reason behind this difference can be found in Fig. 11 and Fig. 12. Fig. 12(a) shows that RW-FCL has a higher hit rate than FP-FCL at cache size 80%. On the other hand, Fig. 11(a) shows that for cache size of 70% to 80% the GC cost increase is steeper for FP-FCL than for RW-FCL. These results imply that, for RW-FCL, the positive effect of caching more data is greater than the negative effect of increased GC cost at 80% cache size, and vice versa for FP-FCL. These differences in positive and negative effect relations for FP-FCL and RW-FCL result in different minimal points.

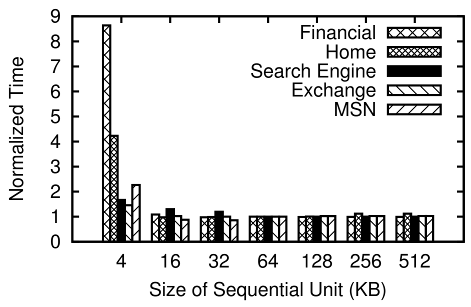
From the results of the MSN trace shown in Fig. 10(b), we observe that FP-FCL and RW-FCL have almost identical response times. This is because they have almost the same hit rate curves, which means that discriminating read and write requests has no performance benefit for the MSN trace. The minimal points for FP-FCL and RW-FCL are at cache size 80% for this trace.

As with the small scale workloads, Fig. 13 shows how OP-FCL adjusts the cache and OPS sizes according to the reference pattern for the large scale workloads. Initially, the cache size starts to increase as we start with an empty cache. Then, we see that the scheme stabilizes with OP-FCL dynamically adjusting the caching space and OPS sizes to their optimal values.

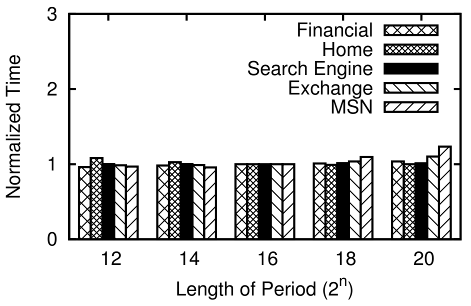
6.3 Effect on lifetime of SSDs

Now let us turn our attention to the effect of OP-FCL on the lifetime of SSDs. Generally, block erase count, which is affected by the wear-levelling technique used by the SSDs, directly corresponds to SSD lifetime. Therefore, we measure the average erase counts of flash memory blocks for all the workloads, and the results are shown in Fig. 14. With the exception of the Search Engine, we see that, for FP-FCL and RW-FCL, the average erase count is low when caching space is small. As caching space becomes larger, the average erase count increases only slightly until the caching space reaches around 70%. Beyond that point, the erase count increases sharply as OPS size becomes small and GC cost rises. In contrast, OP-FCL has a low average erase count drawn as a horizontal line in Fig. 14.

In contrast to the other traces, the average erase count for the Search Engine trace is rather unique. First, the overall average erase count is noticeably lower than that of the other traces. Also, instead of a sharp increase observed for the other traces, we first see a noticeable drop as the cache size approaches 80%, before a sharp increase. The reason behind this is that 99% of the Search Engine trace are read requests and the footprint is so



(a) Effect of sequential unit size



(b) Effect of period length

Figure 15: Sensitivity analysis of sequential unit size and period length on OP-FCL performance

huge that the cache hit rate continuously increases almost linearly with larger caches as shown in Fig. 8(c). This continuous increase in hit rate continuously reduces new writes resulting in reduced garbage collection, and then eventually to reduced block erases. Beyond the 80% point, block erases increase because GC cost increases sharply as the OPS becomes smaller.

6.4 Sensitivity analysis

In this subsection, we present the effect on the choice of the sequential unit size and the length of the period on the performance of OP-FCL. The results for all the workloads are reported relative to the parameter settings used for all the results presented in the previous subsections: the sequential unit size of 128 and period length of 2^{16}.

Recall that the sequential unit size determines the consecutive request size that the Sequential I/O Detector regards as being sequential, and that these requests are sent directly to the HDD. Fig. 15(a) show the effect of the sequential unit size. When the sequential unit size is 4 KB, OP-FCL performs very poorly. This is because too many requests are being considered to be sequential and are sent directly to the HDD. However, when the sequential unit size is between 16 KB $\sim$ 512 KB, OP-FCL shows similar performance showing that performance is relatively insensitive to the parameter of choice.

Fig. 15(b) shows the performance of OP-FCL as the length of the period is varied from 2^{12} to 2^{20} requests. Overall, the performance is stable. The Home trace performance deteriorates somewhat for periods of 2^{14} and below, with worse performance as the period shortens. The reason behind this is that the workload changes frequently as observed in Fig. 9. As a result, by the time OP-FCL adapts to the results of the previous period, the new adjustment becomes stale, resulting in performance reduction. We also see that performance is relatively consistent and best for periods between 2^{14} to 2^{16}. For periods beyond 2^{18}, OP-FCL performance deteriorates slightly. As the period increases to 2^{20}, performance of the Exchange and MSN traces start to degrade. This is because the change in the workload spans a relatively large range compared to those of small scale workloads as shown in Fig. 13. For this reason, OP-FCL of longer periods is not dynamic enough to reflect these workload changes effectively. Overall though, we find that for a relatively broad range of periods performance is consistent.

7 Conclusions

NAND flash memory based SSDs are being used as non-volatile caches in hybrid storage solutions. In flash based storage systems, there exists a trade-off between increasing the benefits of caching data and increasing the benefit of reducing the update cost as garbage collection cost is involved. To increase the former, caching space, which is cache space that holds normal data, must be increased, while to increase the latter, over-provisioned space (OPS) must be increased. In this paper, we showed that balancing the caching space and OPS sizes has a significant impact on the performance of hybrid storage solutions. For this balancing act, we derived cost models to determine the optimal caching space and OPS sizes, and proposed a scheme that dynamically adjusts sizes of these spaces. Through experiments we show that our dynamic scheme performs comparatively to the off-line optimal fixed partitioning scheme. We also show that the lifetime of SSDs may be extended considerably as the erase count at SSDs may be reduced.

Many studies on non-volatile cache have focussed on cache replacement and destaging policies. As a miss at the flash memory cache leads to HDD access, it is critical that misses be reduced. When misses do occur at the write cache, intelligent destaging should help ameliorate miss effects. Hence, we are currently focusing our efforts on developing better cache replacement and destaging policies, and in combining these policies with our cache partitioning scheme. Another direction of research that we are pursuing is managing the flash memory cache layer to tune SSDs to trade-off between performance and lifetime.

8 Acknowledgments

We would like to thank our shepherd Margo Seltzer and anonymous reviewers for their insight and suggestions for improvement. This work was supported in part by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. R0A-2007-000-20071-0), by the Korea Science and Engineering Foundation (KOSEF) grant funded by the Korea government (MEST) (No. 2009-0085883), 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] AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J. D., MANASSE, M., AND PANIGRAHY, R. Design Tradeoffs for SSD Performance. In *Proc. of USENIX ATC* (2008), pp. 57–70.

[2] BUCY, J. S., SCHINDLER, J., SCHLOSSER, S. W., AND GANGER, G. R. DiskSim 4.0. *http://www.pdl.cmu.edu/DiskSim/*.

[3] CHEN, F., JIANG, S., AND ZHANG, X. SmartSaver: Turning Flash Drive into a Disk Energy Saver for Mobile Computers. In *Proc. of ISLPED* (2006), pp. 412–417.

[4] CHEN, F., KOUFATY, D. A., AND ZHANG, X. Hystor: Making the Best Use of Solid State Drives in High Performance Storage Systems. In *Proc. of ICS* (2011), pp. 22–32.

[5] DEBNATH, B., SENGUPTA, S., AND LI, J. ChunkStash: Speeding Up Inline Storage Deduplication using Flash Memory. In *Proc. of ATC* (2010).

[6] FIU TRACE REPOSITORY. *http://sylab.cs.fiu.edu/projects/iodedup*.

[7] GUPTA, A., KIM, Y., AND URGAONKAR, B. DFTL: A Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings. In *Proc. of ASPLOS* (2009), pp. 229–240.

[8] HONG, S., AND SHIN, D. NAND Flash-Based Disk Cache Using SLC/MLC Combined Flash Memory. In *Proc. of SNAPI* (2010), pp. 21–30.

[9] HU, X.-Y., ELEFTHERIOU, E., HAAS, R., ILIADIS, I., AND PLETKA, R. Write Amplification Analysis in Flash-based Solid State Drives. In *Proc. of SYSTOR* (2009).

[10] KANG, J.-U., JO, H., KIM, J.-S., AND LEE, J. A Superblock-based Flash Translation Layer for NAND Flash Memory. In *Proc. of EMSOFT* (2006), pp. 161–170.

[11] KGIL, T., ROBERTS, D., AND MUDGE, T. Improving NAND Flash Based Disk Caches. In *Proc. of ISCA* (2008), pp. 327–338.

[12] KIM, J., KIM, J. M., NOH, S. H., MIN, S. L., AND CHO, Y. A Space-Efficient Flash Translation Layer for CompactFlash Systems. *IEEE Trans. on Consumer Electronics 48*, 2 (2002), 366–375.

[13] 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 *Proc. of OSDI* (2000).

[14] KIM, S.-H., JUNG, D., KIM, J.-S., AND MAENG, S. HeteroDrive: Reshaping the Storage Access Pattern of OLTP Workload Using SSD. In *Proc. of IWSSPS* (2009), pp. 13–17.

[15] KIM, Y., GUPTA, A., URGAONKAR, B., BERMAN, P., AND SIVASUBRAMANIAM, A. HybridStore: A Cost-Efficient, High-Performance Storage System Combining SSDs and HDDs. In *Proc. of MASCOTS* (2011), pp. 227–236.

[16] KOLLER, R., AND RANGASWAMI, R. I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance. In *Proc. of FAST* (2010).

[17] KWON, H., KIM, E., CHOI, J., LEE, D., AND NOH, S. H. Janus-FTL: Finding the Optimal Point on the Spectrum Between Page and Block Mapping Schemes. In *Proc. of EMSOFT* (2010), pp. 169–178.

[18] LEE, H. J., LEE, K. H., AND NOH, S. H. Augmenting RAID with an SSD for Energy Relief. In *Proc. of HotPower* (2008).

[19] LEE, S.-W., PARK, D.-J., CHUNG, T.-S., LEE, D.-H., PARK, S., AND SONG, H.-J. A Log Buffer-Based Flash Translation Layer Using Fully-Associative Sector Translation. *ACM Trans. on Embedded Computer Systems 6*, 3 (2007).

[20] MENON, J. A Performance Comparison of RAID-5 and Log-Structured Arrays. In *Proc. of HPDC* (1995).

[21] NARAYANAN, D., DONNELLY, A., THERESKA, E., ELNIKETY, S., AND ROWSTRON, A. Everest: Scaling Down Peak Loads Through I/O Off-Loading. In *Proc. of OSDI* (2008), pp. 15–28.

[22] NARAYANAN, D., THERESKA, E., DONNELLY, A., ELNIKETY, S., AND ROWSTRON, A. Migrating Server Storage to SSDs: Analysis of Tradeoffs. In *Proc. of EuroSys* (2009), pp. 145–158.

[23] PARK, C., CHEON, W., KANG, J., ROH, K., CHO, W., AND KIM, J.-S. A Reconfigurable FTL (Flash Translation Layer) Architecture for NAND Flash-Based Applications. *ACM Trans. on Embedded Computer Systems 7*, 4 (2008).

[24] PARK, J., LEE, H., HYUN, S., KOH, K., AND BAHN, H. A Cost-aware Page Replacement Algorithm for NAND Flash Based Mobile Embedded Systems. In *Proc. of EMSOFT* (2009), pp. 315–324.

[25] PATTERSON, R. H., GIBSON, G. A., GINTING, E., STODOLSKY, D., AND ZELENKA, J. Informed Prefetching and Caching. In *Proc. of SOSP* (1995), pp. 79–95.

[26] PRITCHETT, T., AND THOTTETHODI, M. SieveStore: A Highly-Selective, Ensemble-level Disk Cache for Cost-Performance. In *Proc. of ISCA* (2010), pp. 163–174.

[27] ROSENBLUM, M., AND OUSTERHOUT, J. K. The Design and Implementation of a Log-Structured File System. *ACM Trans. on Computer Systems 10*, 1 (1992), 26–52.

[28] SAXENA, M., AND SWIFT, M. M. FlashVM: Virtual Memory Management on Flash. In *Proc. of ATC* (2010).

[29] SCHINDLER, J., SHETE, S., AND SMITH, K. A. Improving throughput for small disk requests with proximal I/O. In *Proc. of FAST* (2011).

[30] SEAGATE MOMETUS ®XT. *http://www.seagate.com/www/en-us/products/laptops/laptop-hdd*.

[31] SHIM, H., SEO, B.-K., KIM, J.-S., AND MAENG, S. An Adaptive Partitioning Scheme for DRAM-based Cache in Solid State Drives. In *Proc. of MSST* (2010).

[32] SOUNDARARAJAN, G., PRABHAKARAN, V., BALAKRISHNAN, M., AND WOBBER, T. Extending SSD Lifetimes with Disk-Based Write Caches. In *Proc. of FAST* (2010).

[33] UMASS TRACE REPOSITORY. *http://traces.cs.umass.edu*.

[34] UNDERSTANDING THE FLASH TRANSLATION LAYER (FTL) SPECICATION. *Intel Corporation*, 1998.

[35] WANG, W., ZHAO, Y., AND BUNT, R. HyLog: A High Performance Approach to Managing Disk Layout. In *Proc. of FAST* (2004), pp. 145–158.

Lifetime Management of Flash-Based SSDs Using Recovery-Aware Dynamic Throttling

Sungjin Lee, Taejin Kim, Kyungho Kim*, and Jihong Kim

Seoul National University, Korea
{*chamdoo, taejin1999, jihong*}*@davinci.snu.ac.kr*
Samsung Electronics, Korea
kyungho21.kim@samsung.com

Abstract

NAND flash-based solid-state drives (SSDs) are increasingly popular in enterprise server systems because of their advantages over hard disk drives such as higher performance and lower power consumption. However, the limited and unpredictable lifetime of SSDs remains to be a serious obstacle to wider adoption of SSDs in enterprise systems. In this paper, we propose a novel recovery-aware dynamic throttling technique, called READY, which guarantees the SSD lifetime required by the enterprise market while exploiting the self-recovery effect of floating-gate transistors. Unlike a static throttling technique, the proposed technique makes throttling decisions dynamically based on the predicted future write demand of a workload so that the required SSD lifetime can be guaranteed with less performance degradation. The proposed READY technique also considers the self-recovery effect of floating-gate transistors which improves the endurance of SSDs, enabling to guarantee the required lifetime with less write throttling. Our experimental results show that the proposed READY technique can improve write performance by 4.4x with less variations on the write time over the existing static throttling technique while guaranteeing the required SSD lifetime.

1 Introduction

NAND flash memory has been widely used in mobile embedded systems like mobile phones, MP3 players, and laptop computers because of its low-power consumption, high mobility, and high performance. Recently, as the price-per-byte of NAND flash memory is falling, NAND flash-based solid-state drives (SSDs) are increasingly popular in enterprise servers as well, replacing hard disk drives. However, the poor write endurance of NAND flash memory is still regarded as a main barrier for a wide adoption of flash-based SSDs in the enterprise market. In order for SSDs to be broadly adopted in the enterprise environment, two key problems on the SSD lifetime need to be addressed properly.

The first problem is that the *endurance* of flash devices is rapidly decreasing. The endurance of flash-based SSDs is directly related to the number of program/erase (P/E) cycles allowed to memory cells, which are made from floating-gate transistors. Due to the charge trapping characteristic of a floating-gate transistor [1, 2], NAND flash memory is gradually impaired as the number of P/E cycles increases and becomes unreliable beyond a maximum number of P/E cycles. As the semiconductor process is scaled down and with multi-level cell (MLC) technology, the endurance of a floating-gate transistor is significantly degraded. For example, the maximum number of P/E cycles of single-level cell (SLC) flash memory fabricated in a 70 nm process is about 100K P/E cycles. For 2-bit MLC flash memory fabricated in the 2x nm process, the maximum number of P/E cycles decreases to 3K P/E cycles [3, 4, 5] while, for 3-bit MLC flash memory, this number is only a few hundred cycles [6].

The second problem is the *unpredictable lifetime* of flash devices. Since the endurance of SSDs is dependent upon the number of P/E cycles, the SSD lifetime is determined by extra data written by garbage collection and wear-leveling as well as by the number of bytes written by applications. This means that, unlike HDDs, the SSD lifetime is a function of a workload. Therefore, even if the endurance of SSDs seems sufficient, the lifetime of SSDs strongly depends on the write intensiveness of the workload. For example, SSDs may achieve the required lifetime if a small number of write requests are required from applications. On the other hand, the same SSDs will fail much earlier if they are used in a write intensive environment. In particular, as cost-effective MLC-based SSDs are becoming popular in the enterprise market where write requests are intensive [7, 8], it is a challenge to guarantee a minimum SSD lifetime of 3-5 years, which enterprise customers often require [9].

In this paper, we overcome these technical difficulties by proposing a **re**covery-**a**ware **dy**namic throttling technique, called READY. A basic concept of READY is to throttle write performance by adding throttling delays to write requests, so as to guarantee the required SSD lifetime. With dynamic throttling, the IOPS and bandwidth of SSDs is reduced to a certain extent. From the application prospective, applications' execution times are increased as if they run on top of a slower device. As a result, the amount of write traffic sent to a storage device is reduced, lessening the wearing-rate of SSDs.

The dynamic throttling technique inevitably reduces the overall write performance. In order to mitigate performance degradation, we carefully determine throttling delay by predicting future write demands and distribute the predicted delay over the entire SSD lifetime so that better write response time can be obtained with less variations on the response time. In addition, the proposed dynamic throttling technique takes into account the self-recovery characteristic of a floating-gate transistor. Because of the physical characteristics of NAND flash memory, the damage caused by repetitive P/E cycles can be partially recovered during the idle period between two consecutive P/E cycles, improving the endurance of a floating-gate transistor [1, 2, 10, 11, 12]. By considering the endurance improvement by the self-recovery effect, the proposed READY technique can be more optimistic on the total number of data written, thus employing a smaller throttling delay. Our evaluation results show that the proposed throttling technique improves the average write response time by 4.4x with less variations over an existing static throttling technique while guaranteeing the SSD lifetime.

This paper is organized as follows. In Section 2, we briefly explain the endurance characteristics of NAND flash memory. Section 3 describes the proposed recovery-aware dynamic throttling technique in detail. In Section 4, we evaluate the effectiveness of the proposed recovery-aware dynamic throttling technique using enterprise benchmarks. Section 5 describes related work on improving the SSD endurance. Finally, Section 6 concludes with summary and directions for future work.

2 Endurance Characteristics of Flash Memory

In NAND flash memory, program/erase (P/E) operations inevitably cause damage to floating-gate transistors, reducing the overall endurance of memory cells. At the device level, memory cells are gradually worn out as charges get trapped in the interface and oxide layers of a floating-gate transistor during P/E cycles. This charge trapping increases the threshold voltage of a floating-gate, which indicates a logical bit value of a cell, and the cell becomes unreliable when the threshold voltage is higher than a certain voltage margin, e.g., 0.65V for MLC flash memory [1]. According to [1, 10], the increase, δV_{trap}, in a threshold voltage because of charge trapping approximately scales with P/E cycles in a power-law fashion as follows:

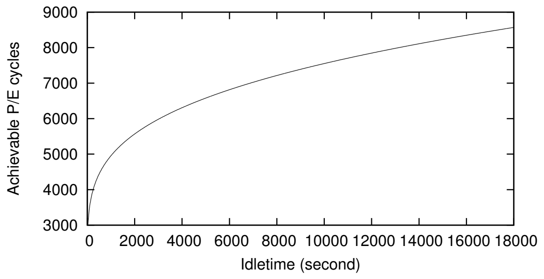


Figure 1: The achievable number of P/E cycles depending on the different idle times.

$$\delta V_{trap} = A_{it} \cdot N^{0.62} + B_{ot} \cdot N^{0.3}, \tag{1}$$

where N is the number of P/E cycles. A_{it} and B_{ot} are constant and set to 2.97×10^{-3} and 2.0×10^{-2}, respectively. Usually, NAND flash memory vendors do not reveal important parameters for their recent products. Thus, in this work, A_{it} and B_{ot} for 20 nm MLC flash memory are obtained by scaling up values for 90 nm MLC flash memory, which are available to the public, so that the number of P/E cycles approximately matches 3K at the point where δV_{trap} is 0.65V.

A floating-gate transistor also has a self-recovery property which heals the damage of a cell by detrapping charges captured in the oxide of a cell. This recovery (or detrapping) process occurs during the idle time between P/E cycles on the same cell, and its effect in general increases as the logarithm of the idle time, i.e., detrapping $\propto \ln(t)$, where t is the length of the idle time. According to [1, 10, 13], the decrease, δV_{detrap}, in a threshold voltage due to charge detrapping can be expressed as follows:

$$\delta V_{detrap} = C_e \cdot \delta V_{trap} \cdot ln(\frac{t}{t_0}), \tag{2}$$

where C_e is a recovery efficiency and set to 5.63×10^{-2} according to [2]. t_0 is 1 hour.

Besides the length of the idle time, there are other factors that affect the cell recovery, such as an external temperature and a programmed threshold voltage. In this work, the temperature is assumed to be a room temperature 25°C because the external ambient temperature of a storage device is typically maintained at the room temperature [14]. The programmed threshold voltage is not taken into account in this study because its effect on the damage recovery is relatively negligible.

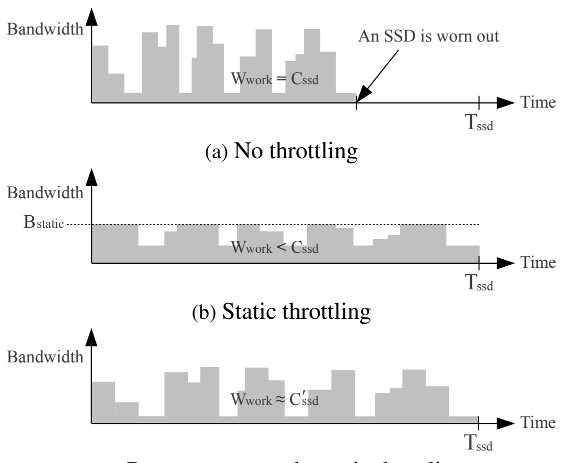


Figure 2: A comparison of three difference throttling policies: no throttling, static throttling, and recovery-aware dynamic throttling.

According to [1], the effective increase, δV_{th}, in a threshold voltage can be expressed as follows:

$$\delta V_{th} = \delta V_{trap} - \delta V_{detrap}. \quad (3)$$

Based on Eq. (3), we have plotted the achievable P/E cycles of 20 nm MLC flash memory in Figure 1, depending on the average idle times between two consecutive P/E cycles on the same block. The maximum P/E cycles without the recovery effect are 3K. As expected, the achievable P/E cycles are gradually increased in proportional to the length of the idle time. Note that recent studies that measured the effective P/E cycles of real NAND flash parts also reported that the endurance of NAND flash memory is higher than P/E cycles in datasheets [15, 16].

The detrapping phenomenon of a floating-gate transistor has a positive effect on improving the endurance (or increasing P/E cycles) of flash-based SSDs. However, most studies use a fixed number of P/E cycles, e.g., 3K P/E cycles, provided by flash manufacturers as a primary factor to manage the lifetime of SSDs. Therefore, the benefit of the damage recovery is not fully utilized. Unlike other studies, our recovery-aware dynamic throttling technique takes advantage of the self-recovery effect in managing the lifetime of SSDs to lessen the performance penalty caused by write throttling.

3 Recovery-Aware Dynamic Throttling

In this section, we describe the proposed recovery-aware dynamic throttling technique. We first introduce the need for dynamic throttling in flash-based SSDs using a simple motivational example and then explain the main functions of the proposed throttling technique in detail.

3.1 Basic Idea

Figure 2 shows a motivational example of dynamic throttling in SSDs. The maximum number, C_{ssd}, of bytes that can be written to the SSD is proportional to the SSD capacity and the number of P/E cycles allowed to each block. C_{ssd} is thus easily calculated with the following equation: SSD capacity × P/E cycles [17]. For example, if the SSD capacity is 128 GB and the number of P/E cycles is 3K, C_{ssd} becomes 375 TB. Suppose that a lifetime, T_{ssd}, to be guaranteed is 1.5768 $\cdot 10^8$ seconds, i.e., 5 years. In the example of Figure 2(a) which does not use write throttling, the required lifetime cannot be satisfied because the number, W_{work}, of bytes written to the SSD exceeds C_{ssd} before T_{ssd}.

To ensure the lifetime warranty of the SSD, some SSD vendors recently have started to adopt *static throttling* [18, 19], which is shown in Figure 2(b). Static throttling guarantees the required lifetime by limiting the maximum bandwidth of the SSD to a certain fixed value, which is denoted by B_{static}. Static throttling determines the value of B_{static} based on the assumption of the worst case scenario where the number of bytes written per second is always larger than C_{ssd}/T_{ssd}. In this case, B_{static} must be fixed to C_{ssd}/T_{ssd} to ensure the required lifetime. The drawback of this approach is that it is likely to underutilize the maximum endurance of the SSD, i.e., $W_{work} < C_{ssd}$ at T_{ssd}, because of its assumption that the SSD must provide the B_{static} bandwidth although actual workloads may not be that intensive all the time. In addition, due to this conservative assumption, the I/O response time is degraded with static throttling.

In order to overcome the limitation of the static throttling technique, we propose a recover-aware dynamic throttling technique, READY, which is depicted in Figure 2(c). By dynamically throttling write requests according to the characteristics of a workload and the remaining SSD lifetime, the proposed READY technique fully utilizes the given endurance of the SSD up to the maximum, while minimizing performance degradation. READY is also aware of the endurance improvement by the self-recovery characteristic of memory cells. Therefore, the data that can be written to the SSD increase by ΔC_{ssd}, so the maximum number of writable bytes becomes C'_{ssd} (= $C_{ssd} + \Delta C_{ssd}$). This allows us to guarantee the required lifetime with less throttling overheads.

In designing a dynamic throttling policy, we focus on two aspects of the design requirements of SSDs. The first is to determine a throttling delay as low as possible so that W_{work} is close to C'_{ssd} at the time of T_{ssd}. If $W_{work} = C'_{ssd}$ before T_{ssd}, we cannot guarantee the required lifetime as shown in Figure 2(a). If $W_{work} < C'_{ssd}$ at T_{ssd}, write performance significantly deteriorates, underutilizing the available endurance of the SSD

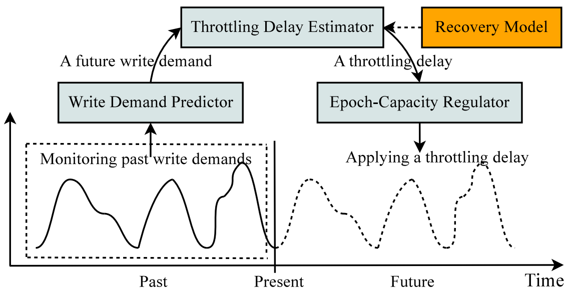


Figure 3: Three main functions of READY.

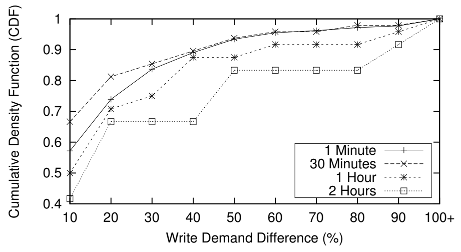
like static throttling as depicted in Figure 2(b). The second is to distribute a throttling delay over every write request as evenly as possible. Otherwise, response time variations can be large, thus lowering the quality of the user experience significantly.

To effectively deal with these design issues, the proposed dynamic throttling technique has been designed with three main functions as shown in Figure 3. The write demand predictor is in charge of predicting future write demands, which indicate the number of bytes that is written to SSDs, by monitoring previous write demands. Once the future demand for writes has been predicted, the throttling delay estimator determines a throttling delay by considering both the future write demand and the remaining lifetime of SSDs. The epoch-capacity regulator throttles write performance by applying a throttling delay to each write request so that the target SSD lifetime will be reached.

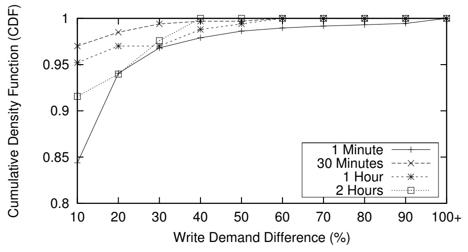
3.2 Estimation of Future Write Demands

In designing a dynamic throttling policy, it is important to estimate the number of bytes that will be written to the SSD in advance because the SSD performance must be throttled properly if the write demand is expected to be too high. The role of the write demand predictor is to predict future write demands by monitoring the previous write demands of a workload.

For this purpose, in READY, the entire lifetime, T_{ssd}, of the SSD is divided into *epochs*. At the beginning of each epoch, the write demand predictor estimates the number of bytes that is to be written during the epoch based on the number of bytes actually written to the SSD during the latest epoch. If the data of w_{i-1} have been written during the $(i-1)$-th epoch, the write demand predictor predicts that the same number of bytes will be written to the SSD during the i-th epoch. That is, $w_i \approx w_{i-1}$. This approach is motivated by previous observations [20] that showed that enterprise workloads often exhibit cyclic behavior with periods between several minutes and several days. Although that work did not address I/O demands in storage devices, it showed that a strong cyclical behavior is frequently observed in



(a) exchange



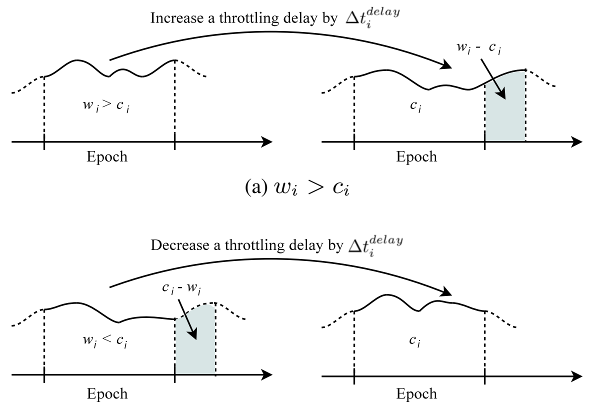
(b) proxy

Figure 4: Write demand differences with different epoch lengths for `exchange` and `proxy`.

enterprise applications. This means that if the length of an epoch is properly decided to include the cyclic period of a workload, the write demand observed in the latest epoch can be used as a factor that indicates future write demands.

To confirm our hypothesis, we have analyzed the characteristic of write demands using enterprise traces. We have compared the difference in write demands between two consecutive epochs while varying the length of an epoch from 1 minute to 2 hours. Our analysis has been performed with several enterprise traces from the MSR-Cambridge and MS-Production traces [21, 22]. Figure 4 shows our investigation results for the two traces, `proxy` and `exchange`. Here, the X-axis represents the write demand difference between the predicted write demand and the actual one in percentage. For example, if the predicted demand is 100 MB and the actual one is 95 MB, the write demand difference between them is 5%. The Y-axis is the cumulative density function (CDF) of the write demand difference of the epochs. The smaller the difference, the better the accuracy of future write demand prediction is.

As shown in Figure 4, when the length of an epoch is decided properly, it is possible to achieve high accuracy in predicting future write demands. In the case of `exchange`, for about 85% of the epochs, the write demand difference of less than 30% is obtained with the epoch length of 30 minutes. For `proxy`, the epoch length of 30 minutes shows the best accuracy in estimating future write demands. This result clearly shows that the epoch-based write demand prediction is useful to estimate future write demands. Note that other methods, such as a moving average, are also applicable for esti-


Increase a throttling delay by Δt_i^{delay}
$w_i - c_i$
$w_i > c_i$
c_i
Epoch
Epoch
Decrease a throttling delay by Δt_i^{delay}
$c_i - w_i$
$w_i < c_i$
c_i
Epoch
Epoch

(a) $w_i > c_i$

(b) $w_i < c_i$

Figure 5: A change in a throttling delay.

mating future write demands.

Since the best epoch length may be different depending on a workload and its characteristic (which changes with time), the proposed READY technique selects the epoch length dynamically adapting to a changing workload. We will discuss this issue in Section 3.5.

3.3 Calculation of Throttling Delay

The throttling delay estimator adaptively changes a throttling delay at every epoch by monitoring the write demand and the remaining SSD lifetime. At the first epoch, i.e., the 0-th epoch, a throttling delay, t_0^{delay}, is set to 0. Then, at the beginning of each i-th epoch, the delay estimator increases or decreases a throttling delay, t_i^{delay}, based on the expected write demand and the capacity of each epoch. The expected write demand indicates the number, w_i, of bytes that is supposed to be written during the i-th epoch. The capacity of an epoch is the number, c_i, of bytes allowed to be written during the i-th epoch. In this work, w_i is equal to the number, w_{i-1}, of bytes written during the $(i-1)$-th epoch under the assumption of $w_i \approx w_{i-1}$. The capacity, c_i, of the i-th epoch is determined by dividing the remaining capacity, C_r, of the SSD by the number of remaining epochs. Here, the remaining capacity, C_r, represents the number of bytes that can be written to the SSD until it becomes unreliable.

If w_i is equal to c_i, we don't need to change a throttling delay for the i-th epoch. Therefore, t_i^{delay} is the same as t_{i-1}^{delay}, which is the throttling delay of the $(i-1)$-th epoch. However, if w_i is larger than c_i as shown in Figure 5(a), it is necessary to increase a throttling delay because the data to be written to the SSD are expected to be larger than the capacity allocated to the epoch. The increase, Δt_i^{delay}, in a throttling delay can be expressed as follows:

$$\Delta t_i^{delay} = t_{epoch} \cdot \left(\frac{w_i}{c_i} - 1\right) \Big/ n \quad \text{if } w_i > c_i, \qquad (4)$$

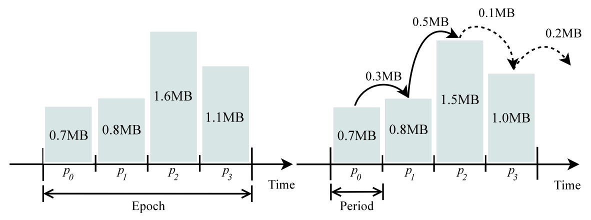
where n is the number of pages allowed to be written to the SSD during the i-th epoch, i.e., c_i/page size, and t_{epoch} is the epoch length. To make the data written during the i-th epoch equal to c_i, $(w_i - c_i)$ of the data must be delayed to the next epoch as shown in Figure 5(a). The total time required to delay $(w_i - c_i)$ of the data can be approximated as $t_{epoch} \cdot (w_i/c_i - 1)$. In our dynamic throttling policy, a throttling delay is equally distributed to each page write (refer to Section 3.4), so Δt_i^{delay} can be obtained by dividing the total throttling delay by n. Finally, a throttling delay, t_i^{delay}, for the i-th epoch is determined as follows: $t_i^{delay} = t_{i-1}^{delay} + \Delta t_i^{delay}$.

If w_i is smaller than c_i as shown in Figure 5(b), it means that the write requests were not intensive enough to wear out the device before the required lifetime or they were too throttled during the previous epoch. Therefore, the throttling delay may be reduced so that more data can be written to the SSD. The decrease, Δt_i^{delay}, in a throttling delay can be expressed as follows:

$$\Delta t_i^{delay} = t_{epoch} \cdot \left(\frac{c_i}{w_i} - 1\right) \Big/ n \quad \text{if } w_i < c_i. \qquad (5)$$

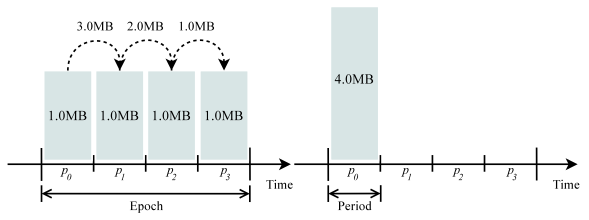
To increase the number of bytes to be written to the SSD by $(c_i - w_i)$ during the i-th epoch, a throttling delay, t_i^{delay}, for the i-th epoch is reduced by Δt_i^{delay} as follows: $t_i^{delay} = t_{i-1}^{delay} - \Delta t_i^{delay}$. In the case of $t_{i-1}^{delay} < \Delta t_i^{delay}$, t_i^{delay} is 0. This means that it is not necessary to apply a throttling delay because the required lifetime can be guaranteed without write throttling.

Until now, we assumed that the number of P/E cycles is fixed to a certain number. The achievable P/E cycles, however, can be increased depending on the amount of the idle time between two consecutive P/E cycles in a certain block because of the self-recovery effect of memory cells. In order to exploit this endurance improvement, the throttling delay estimator first estimates the number of achievable P/E cycles at the beginning of each epoch, using the damage and recovery model mentioned in Section 2. In this work, the number of achievable P/E cycles is estimated, using the average idle time of every block in the SSD. The idle time is actually somewhat different among blocks. However, this difference is not significant because the wear-leveler of the SSD makes the P/E cycles of all available blocks evenly distributed. Therefore, the average idle time can be used as a useful parameter to estimate the overall endurance improvement of the SSD. The estimator then calculates the remaining capacity, C_r, based on the achievable P/E cycles and distributes it to the remaining epochs. Since the number of P/E cycles is increased due to the recovery effect, the capacity, c_i, of each epoch is also increased, allowing more data to be written to the SSD with less throttling delays.



(a) Without epoch-capacity enforcement (b) With pessimistic epoch-capacity enforcement

Figure 6: An example of epoch-capacity enforcement. A solid line indicates the unused capacity forwarded to the next period and a dashed line represents the data delayed to the next period or epoch.



(a) With pessimistic epoch-capacity enforcement (b) With optimistic epoch-capacity enforcement

Figure 7: A comparison of the pessimistic and optimistic epoch-capacity enforcement policies when the 4 MB data are written during the period p_0.

3.4 Enforcement of Epoch Capacity

Once a throttling delay is decided, we throttle SSD performance by distributing throttling delays across every write as evenly as possible. This regulation policy is beneficial in minimizing response time variations, but it cannot guarantee the required lifetime if write demand prediction fails and unexpectedly high write traffic comes from the host. To resolve this problem, it is necessary to adopt an epoch-capacity enforcement policy, which prevents more data than the epoch capacity from being written to the SSD.

One of the easiest ways to enforce the epoch capacity is to stop writing if the epoch capacity is likely to be exhausted before the epoch ends. We call such a regulation strategy the *pessimistic* epoch-capacity enforcement policy. The pessimistic policy divides one epoch into *periods* whose lengths are 1 second each and then distributes the capacity of an epoch to all periods evenly. If more data than the period capacity were requested to write, the epoch-capacity regulator stops writing so that overflowed requests are to be written in the next period. If there is an unused capacity in the current period, the regulator reallocates it to the next period so that it can be used during the next period. This period-based capacity regulation allows us to maintain the minimum write throughput when there is unexpectedly high write traffic. If we stop writing after the epoch capacity is exhausted, the SSD cannot write any data until the epoch ends with significant performance degradation. Figure 6 compares the situations where no epoch-capacity enforcement policy is used and the pessimistic policy is used. Here, we assume that the epoch capacity is 4 MB and the number of periods is 4. As shown in Figure 6(a), the 4.2 MB data are written to the SSD without epoch-capacity enforcement. With pessimistic epoch-capacity enforcement, the maximum number of bytes written to the SSD is limited to 4.0 MB as shown in Figure 6(b).

The weakness of the pessimistic policy is that it does not efficiently handle a bursty I/O pattern which writes a large number of data within a relatively short period. Figure 7(a) shows how the pessimistic policy behaves under a bursty write request. We assume that the capacity of an epoch is 4 MB and the number of periods is 4. Consider that the 4 MB data are requested during the period p_0 while no write requests are issued during the periods $p1$, $p2$, and $p3$. In this example, the pessimistic policy throttles write requests for every period except for p_3 because the requested data always exceed the maximum capacity of the period. However, since the total number of bytes written during the epoch is equal to 4 MB, throttling for the periods, p_0, p_1, and p_2, is, in fact, unnecessary.

We resolve this overly restrictive throttling behavior for bursty write requests by proposing the *optimistic* epoch-capacity enforcement policy. Our optimistic policy maintains a relatively small amount of the spare capacity for each epoch and forcibly throttles write performance only when both the capacity of a period and the spare capacity are exhausted. Figure 7(b) shows an example of the optimistic policy with the same scenario shown in Figure 7(a). Here, we assume that the spare capacity is set to 4 MB. As shown in Figure 7(b), unnecessary throttling can be completely avoided with the optimistic epoch-capacity enforcement policy.

The spare capacity must be carefully chosen. Suppose that the spare capacity is unlimited and there is unexpectedly high write traffic. In that case, READY borrows as much capacity as possible from future epochs without limitation and then uses it up. If unexpected write demands frequently occur and write demands are gradually increasing, the SSD is worn out before the required lifetime. On the other hand, if the spare capacity is too small, unnecessary throttling with a bursty I/O pattern would be frequently observed due to the lack of spare capacity. In this work, the spare capacity is empirically set to 10% of the remaining capacity, C_r, of the SSD. This capacity is sufficient enough to avoid unnecessary throttling in real-world traces. Furthermore, since the spare capacity is limited to 10% of C_r, the worn-out of SSDs before the

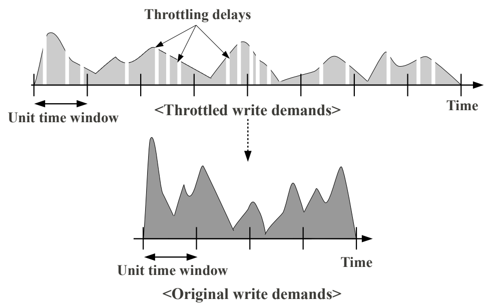


Figure 8: Reconstruction of write demand distribution.

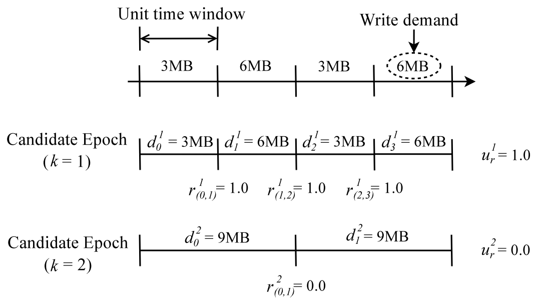


Figure 9: An overall procedure of epoch length selection.

target lifetime never occurs.

Suppose that the spare capacity is 10% and there are n epochs. The capacity of each epoch is c_0, ..., c_{n-1}, respectively. Note that $c_0 = ... = c_{n-1} = C_r/n$ as mentioned in Section 3.3. The spare capacity for the 0-th epoch is $(c_1 + ... + c_{n-1}) \cdot 0.1$, and thus the total capacity that can be written during the 0-th epoch is $c_0 + (c_1 + ... + c_{n-1}) \cdot 0.1$. If n is 3 and C_r is 3 MB, c_0 is 1 MB and the spare capacity is 0.2 MB. If the data of less than c_0 have been written during the 0-th epoch, the remaining capacity, C_r, of the SSD after the 0-th epoch is equally distributed to the remaining epochs and then the spare capacity is determined by $(c_2 + ... + c_{n-1}) \cdot 0.1$ for the 1-st epoch. If the spare capacity, however, is partially used during the 0-th epoch, then c_1, ... , c_{n-1} are reduced to 90% of their original capacities and only the unallocated capacity is used as the spare capacity. For example, in the above example, if the data of 1.1 MB have been written during the 0-th epoch, c_1 and c_2 are 0.9 MB and the spare capacity becomes 0.1 MB in the 1-st epoch. This capacity assignment policy makes the throttling delay estimator slightly increase a throttling delay with a smaller epoch capacity. The overused capacity is accordingly reclaimed during the remaining epochs. If the spare capacity is used up during the 0-th epoch, the pessimistic policy is used with the reduced epoch capacity, i.e., 0.9 MB, and no spare capacity. This means that performance degradation caused by the depletion of the spare capacity is 10% in the worst case.

3.5 Epoch Length Selection

The length of an epoch must be carefully decided. If the epoch length is chosen improperly, the difference in write demands between epochs becomes large. Since a throttling delay is determined by the write demand of the previous epoch, the incorrect epoch length can make a large fluctuation in the overall I/O response time of the SSD. To determine the proper epoch length, we monitor write demands of a workload and find repeated cycles that show similar write demands. We then choose that cycle as the epoch length.

To realize this in READY, we collect information about write demands, i.e., the number of bytes written per unit-time window, at runtime. The write demands collected here, however, include throttling delays that distort the actual write demands of applications. Therefore, it is necessary to reconstruct write demand distribution when throttling is not applied. We estimate this original write demands by rebuilding unit-time windows without throttling delays as shown in Figure 8.

To find the proper epoch length, we use a simple approach that attempts to find the best epoch candidate, which exhibits the smallest fluctuation in write demands, by creating and evaluating several candidate epochs with different lengths. Figure 9 shows our approach in choosing an epoch length. We first create a candidate epoch whose length, k, is one unit-time window, i.e., k = 1. We then calculate the write-demand difference ratio of two consecutive epochs i and $i+1$ with the same length. The write-demand difference ratio, $r^k_{(i,i+1)}$, is defined as follows:

$$r^k_{(i,i+1)} = \frac{|d^k_{i+1} - d^k_i|}{d^k_i}, \tag{6}$$

where d^k_i and d^k_{i+1} are the number of bytes written during the epochs i and $i+1$, respectively. For example, in the example of Figure 9, d^1_0 and d^1_1 are 3 MB and 6 MB, respectively, and thus $r^1_{(0,1)}$ = 1.0. We calculate the average write-demand difference ratio, μ^k_r, for all available pairs of two consecutive epochs. In the example of Figure 9, μ^1_r for $r^1_{(0,1)}$, ..., $r^1_{(2,3)}$ becomes 1.0.

We then increase the length of a candidate epoch by one unit-time window and calculate μ^{k+1}_r. We repeat this until the number of epochs with the same length becomes one. Finally, the length of a candidate epoch whose average write-demand difference ratio is the smallest is chosen as the epoch length, t_{epoch}. For example, in Figure 9, μ^k_r is the smallest when k is 2, and thus the new epoch length becomes the length of two unit-time windows. After choosing the new epoch length, READY recalculates a throttling delay using Eq.(4) if dynamic throttling is necessary. The new epoch length is determined under the assumption that there are no throttling delays. The epoch length, t_{epoch}, is thus increased to $t_{epoch} \cdot (w_i/c_i)$

to include delays caused by throttling.

Finding the epoch length may take a relatively long time. To mitigate the computational overhead caused by epoch length selection, the epoch length is recalculated when the write-demand difference ratio between the predicted write demand and the actual one is higher than 0.25 and it occurs three times successively. The length of a unit-time window is also set to 10 minutes to further reduce the computational overhead. In this work, 0.25 is chosen empirically by considering both computational overhead and the accuracy of write demand prediction. However, this number can be further optimized in several ways. For example, the write-demand difference ratio that triggers epoch length recalculation can be adaptively changed depending on the characteristics of a workload. If the difference ratio is always smaller than 0.25, we can reduce this number, e.g., 0.15, to find a better epoch length. On the other hand, if the difference ratio is much larger than 0.25 all the time, it may be better to reduce this number, e.g., 0.35, to avoid useless computational overhead.

4 Experimental Results

In this section, we first describe our experimental settings and explain enterprise benchmarks used for the evaluations in detail. We then analyze the benefit of the proposed READY technique over the static throttling technique in terms of SSD lifetime, response time, and response time variations.

4.1 Experimental Settings

To evaluate the effectiveness of the proposed READY technique, we have performed our evaluations using the DiskSim-based SSD simulator [23]. The flash memory used for the evaluations was based on 2-bit MLC NAND flash memory, and each block was composed of 64 4 KB pages. The page read time and the page write time were 50 μs and 600 μs, respectively, and the block erasure time was 2 ms. The number of P/E cycles allowed to a block was initially set to 3K, but it was changed depending on the length of the idle time based on our recovery model. The target lifetime of the SSD was set to 5 years.

We have implemented the static and dynamic throttling techniques in the SSD simulator, along with the damage and recovery model described in Section 2. The throttling module was implemented between the host interface, e.g., SATA, and the flash translation layer (FTL). The throttling module intercepted write requests destined for the FTL and then applied a throttling delay if it was required for the lifetime guarantee. The FTL employed a page-level address mapping algorithm with a greedy garbage collection policy and used a hot-cold swapping algorithm for wear-leveling [23]. Note that there were no changes at the FTL level for throttling because the throttling module has been designed to operate independently regardless of the underlying FTL algorithms.

Trace	Duration	Data written per hour (GB)	WAF	SSD capacity (GB)
`proxy`	1 week	4.94	1.93	32
`proj`	1 week	2.08	1.62	32
`exchange`	1 day	20.61	2.24	128
`map`	1 day	23.82	1.68	128
`msnfs`	6 hours	18.19	2.26	128

Table 1: A summary of traces used for evaluations.

We compared the performance of five SSD configurations: *NT*, *ST*, *DT*, *READY*$_{\mathrm{PES}}$, and *READY*$_{\mathrm{OPT}}$. *NT* does not use write throttling, so it cannot guarantee the target SSD lifetime if write traffic is very intensive. *ST* and *DT* use static throttling and dynamic throttling, respectively. Note that *DT* uses the optimistic epoch-capacity enforcement policy by default. Both *READY*$_{\mathrm{PES}}$ and *READY*$_{\mathrm{OPT}}$ are different from other configurations in that they take into account the self-recovery effect of memory cells. *READY*$_{\mathrm{PES}}$ uses the pessimistic epoch-capacity enforcement policy, whereas *READY*$_{\mathrm{OPT}}$ employs the optimistic policy.

4.2 Benchmarks

We have chosen two enterprise traces, `proxy` and `proj` from the MSR-Cambridge benchmark [21] and have used three production traces, `exchange`, `map`, and `msnfs`, from the MS-Production benchmark [22]. Table 1 summarizes the traces used for our evaluations. `proxy` and `proj` were recorded for one week. `exchange` and `map` contains 24-hour I/O activities, while `msnfs` was collected for 6 hours. Because of the limited duration of the traces, it was impossible to assess the lifetime guarantee of 5 years with them. For this reason, we performed our evaluations under the assumption that the same I/O pattern is repeated for 5 years.

The write demand is very different depending on the traces. `proxy` and `proj` exhibit a low write demand in comparison with `exchange`, `map`, and `msnfs`. The write amplification factor (WAF), which has a great effect on the write demand, ranges from 1.62 to 2.26 according to the characteristic of I/O references [24]. For the evaluations, the SSD capacity was configured differently depending on the traces so that the lifetime of the SSD is to be a problem. For `proxy` and `proj` with a low write demand, the SSD capacity was set to 32 GB. For `exchange`, `map`, and `msnfs` with a high write demand, the capacity of the SSD was set to 128 GB.

In practice, this capacity planning is carefully decided by customers' requirements. If customers are ready to pay money to obtain a long lifetime and high perfor-

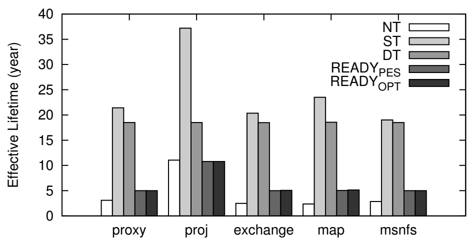


Figure 10: A comparison of effective SSD lifetimes for five traces with different SSD configurations.

mance, an over-provisioned configuration with a larger capacity SSD is the best choice. If customers require a low initial cost, but can manage a relatively high operational cost, a smaller capacity SSD with a shorter target lifetime, i.e., 3 years, is preferred. For customers who want reasonable performance with relatively lower cost and a longer target lifetime, e.g., 5 years, the settings shown in Table 1 may be a better choice.

All the traces used in this work were collected from hard disk drives (HDDs) which exhibit much lower I/O performance than SSDs. Since SSDs increase the overall I/O rate of the storage subsystem by several times [25, 26], the number of bytes written to a storage device during the same time period will be largely increased in comparison with HDDs. That is, 'data written per hour (GB)' shown in Table 1 becomes larger, and thus READY more aggressively throttles write performance because of increased write traffic. Therefore, the SSD capacity in Table 1 is set relatively conservative for systems that use SSDs as a secondary storage device.

4.3 Lifetime Analysis

We first analyze the lifetime of the SSD for five respective traces. Figure 10 shows the effective lifetime of the SSD with different SSD configurations. Here, the effective lifetime is the lifetime which is estimated based on the assumption that the I/O activities of the traces are repeated for 5 years. Note that the self-recovery effect of memory cells is taken into account in estimating the SSD lifetime. As shown in Figure 10, *NT* cannot guarantee the required lifetime of the SSD for all the traces, except for `proj`. In our observation, the write demand of `proj` is not intensive, and thus the SSD can achieve the lifetime more than 5 years without write throttling.

ST and *DT* do not consider the self-recovery effect of floating-gate transistors, and therefore they throttle write performance based on the fixed 3K P/E cycles. Since the P/E cycles of the SSD are increased due to the effect of self-recovery, the effective SSD lifetimes with *ST* and *DT* are much longer than the required lifetime. This means that *ST* and *DT* excessively throttle write performance, underutilizing available P/E cycles of the SSD. This excessive throttling results in poor write performance in comparison with *READY*$_{\text{PES}}$ and *READY*$_{\text{OPT}}$. In particular, *DT* dynamically decides a throttling delay in response to a changing workload. Therefore, *DT* maximizes the utilization of P/E cycles within 3K unlike *ST*. We will discuss this issue in detail with Table 2.

Trace	SSD configuration	C_{ssd}(TB)	C'_{ssd}(TB)	W_{work}(TB)
`proj`	*NT*	93.75	312.6	144.4
	ST		403.4	54.2
	DT		346.9	93.7
	READY$_{\text{PES}}$		312.8	141.0
	READY$_{\text{OPT}}$		312.8	141.0
`exchange`	*NT*	375	949.3	1918.8
	ST		1415.3	348.2
	DT		1387.3	374.4
	READY$_{\text{PES}}$		1077.8	1077.2
	READY$_{\text{OPT}}$		1077.8	1065.6

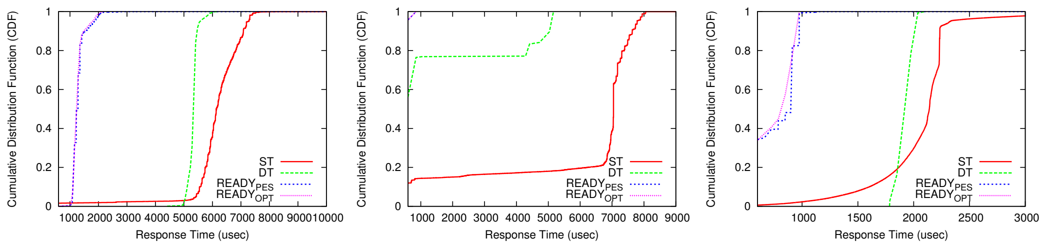
Table 2: The amount of data written for 5 years for two traces, `proj` and `exchange`.

READY$_{\text{PES}}$ takes advantage of the self-recovery effect of memory cells. Therefore, it throttles write performance so that the effective lifetime of the SSD is close to the required lifetime for all the traces. Figure 10 also shows that *READY*$_{\text{OPT}}$ guarantees the required SSD lifetime even though it uses the capacity borrowed from future epochs in advance. This clearly shows that the optimistic epoch-capacity enforcement policy properly manages overused epoch capacity so that the given lifetime is to be satisfied.

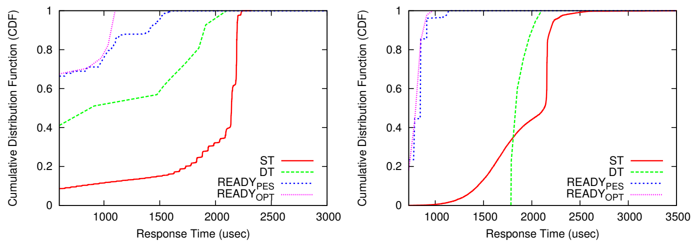
Table 2 analyzes the lifetime of the SSD from the perspective of written data for two traces, `proj` and `exchange`. As mentioned in Section 2, C_{ssd} represents the number of bytes that can be written to the SSD according to the NAND flash memory specification, whereas C'_{ssd} is the total number of writable bytes when the recovery effect is taken into account. W_{work} is the total number of bytes written to the SSD for 5 years.

As expected, *ST* and *DT* throttle write performance so that W_{work} becomes close to C_{ssd}. In particular, in the case of *ST*, W_{work} is about 43% and 8% smaller than C_{ssd} for `proj` and `exchange`, respectively. This is because *ST* excessively throttles write performance, assuming that write requests are always intensive. Unlike *ST*, *DT* dynamically changes a throttling delay according to the write demands of a workload and the remaining lifetime so that W_{work} is close to C_{ssd}, allowing more data to be written to the SSD.

READY$_{\text{PES}}$ and *READY*$_{\text{OPT}}$ fully utilize the endurance improvement offered by the self-recovery effect, making W_{work} close to C'_{ssd} at the target SSD lifetime. In the case of `proj`, since the endurance of the SSD is sufficient enough to guarantee the required 5-year lifetime, throttling is not performed in most cases.



(a) proxy (b) proj (c) exchange



(d) map (e) msnfs

Figure 12: Cumulative distribution functions (CDFs) of write response times.

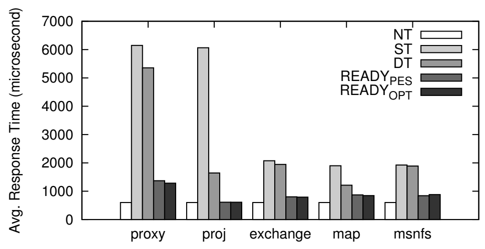


Figure 11: A comparison of average write response times for five traces with different SSD configurations.

4.4 Performance Analysis

To evaluate the performance benefit of the proposed READY technique, we measured the average response time of a page write while running five traces with different SSD configurations. Figure 11 shows the our evaluation results. As expected, *NT* exhibits the best I/O response time among all of the evaluated configurations, but it cannot guarantee the target lifetime as shown in Figure 10 because it does not throttle write performance. The average write response time of *NT* is close to the page access time, i.e., 600 μsec, with little variation.

Both *READY*$_{\text{PES}}$ and *READY*$_{\text{OPT}}$ throttle write requests to meet the required lifetime, so their performance is worse than that of *NT*; they exhibit 1.0x to 2.13x higher write response time than *NT*. In the case of `proj`, *READY*$_{\text{PES}}$ and *READY*$_{\text{OPT}}$ do not reduce write performance because the required lifetime can be satisfied without throttling. Therefore, little performance degradation, which is less than 1.9%, is observed in `proj`. *READY*$_{\text{PES}}$ and *READY*$_{\text{OPT}}$ achieve 2.57x better performance than *DT* on average. This performance benefit mainly comes from the increased P/E cycles of the SSD. Since *READY*$_{\text{PES}}$ and *READY*$_{\text{OPT}}$ are aware of the improvement in SSD endurance, they can assign more capacity to epochs, reducing throttling delays.

DT exhibits 1.7x faster response time over *ST* on average. *DT* determines the epoch capacity periodically based on the remaining lifetime of the SSD and changes a throttling delay so that write requests are properly delayed in response to future write demands. This epoch capacity assignment and throttling delay distribution policy allows us to fully utilize the available endurance of the SSD. On the other hand, *ST* neither considers the remaining lifetime of the SSD nor the characteristic of a workload in making a throttling decision. Instead, *ST* simply throttles write performance by limiting the maximum bandwidth of the SSD. Therefore, *ST* causes many unnecessary throttling delays.

The response time variation is one of the important design issues that must be taken into account in designing throttling algorithms. We compared response time variations between different SSD configurations. Figure 12 shows the cumulative density functions (CDFs) of write response times for five traces. As shown in Figure 12, *ST* shows significant response time variations for all the traces because it forcibly stops writing if throttling is needed. On the other hand, by distributing throttling delays to write requests as evenly as possible, *NT*, *READY*$_{\text{PES}}$, and *READY*$_{\text{OPT}}$ greatly reduce variations on the write response time.

For `exchange`, `msnfs`, and `map`, *READY*$_{\text{PES}}$ incurs relatively high I/O response time variations in comparison with *READY*$_{\text{OPT}}$. *READY*$_{\text{PES}}$ must stop writing data when there are a large number of writes within a short

Traces	proxy	proj	exchange	map	msnfs
Accuracy of write demand prediction (%)	99.9	33.9	80.5	50.9	100

Table 3: Accuracy of write demand prediction.

period. On the other hand, *READY*$_{\text{OPT}}$ uses the optimistic epoch-capacity enforcement policy, so they handle a bursty I/O pattern more efficiently without compulsorily write throttling.

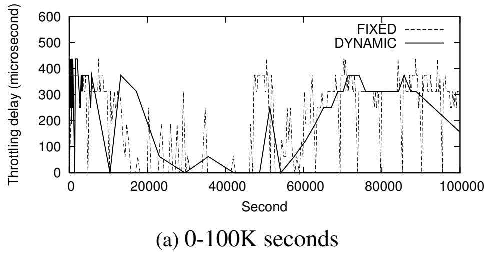
The write response time of *DT* ranges from 600 μsec to several thousand seconds in `map` and `proj` unlike `proxy`, `exchange`, and `msnfs`. The write patterns of `map` and `proj` change greatly with time, and thus the difference in write demands between two consecutive epochs is relatively large. Since a throttling delay for a certain epoch is determined by the write demand of the previous epoch, the difference between throttling delays is accordingly increased in `map` and `proj`. Nevertheless, the response time of *DT* is more stable than *ST*.

We evaluated the accuracy of our epoch length selection method in predicting future write demands. Table 3 shows our evaluation results for five traces. We assume that epoch length detection is accurate if the difference between the prediction write demand and the actual one is smaller than 25%. As shown in Table 3, our method achieves high accuracy for `proxy`, `exchange`, and `msnfs`. The accuracy of write demand prediction, however, is reduced to 50.9% and 33.9% for `map` and `proj`, respectively, due to a high fluctuation in write requests. We expect that the accuracy of epoch length detection may be improved with traces recorded for a longer time.

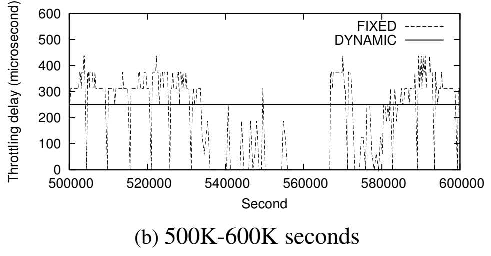
To evaluate the effect of the epoch length selection method on the SSD response time, we compared the changes in throttling delays when the fixed epoch length is used and the epoch length is dynamically determined according to a workload. For the evaluation, we executed the `exchange` trace, which is a 24-hour trace, repeatedly. Figure 13 shows our evaluation result. In this figure, *FIXED* represents *READY*$_{OPT}$ with the fixed epoch length and *DYNAMIC* is the SSD configuration when *READY*$_{OPT}$ uses the proposed epoch length detection method. The fixed epoch length was set to 10 minutes. As shown in Figure 13, even though *READY*$_{OPT}$ generally works well with `exchange`, some variations on throttling delays are observed with *FIXED*. *DYNAMIC* also exhibits variations on response times at the beginning of the execution, but it becomes stable after repeated write demands are detected as shown in Figure 13(b).

4.5 Detailed Analysis

We performed a detailed analysis of different SSD configurations. Figure 14 represents the throughput of the



(a) 0-100K seconds



(b) 500K-600K seconds

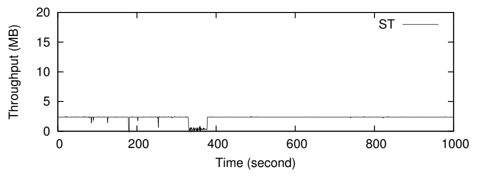
Figure 13: A comparison of throttling delays when the fixed epoch length is used and the epoch length is dynamically determined in the `exchange` trace.

SSD with the different throttling schemes when intense I/Os are being served. As mentioned several times before, *ST* limits the maximum bandwidth of the SSD by a certain level, 2.49 MB/s in `map`. The overall throughput of the SSD is thus greatly deteriorated with *ST* as shown in Figure 14(a). *DT* works better than *ST*. Due to the limited write endurance of the SSD, however, significant performance degradation cannot be avoided with *DT* as depicted in Figure 14(b). *READY*$_{\text{PES}}$ and *READY*$_{\text{OPT}}$ exhibit much higher performance than *ST* and *DT* by exploiting the improved write endurance of the SSD benefited from the self-recovery effect of memory cells. In particular, *READY*$_{\text{OPT}}$ performs better than *READY*$_{\text{PES}}$ when a large number of data are being written to the SSD, e.g., a period of 200 to 350 second in Figure 14(d). Even when write requests are intensively issued, *READY*$_{\text{OPT}}$ writes the requested data to the SSD rather than forcibly throttling the bandwidth of the SSD by using the spare capacity borrowed from future epochs. This allows *READY*$_{\text{OPT}}$ to exhibit better write response time for the traces like `map` which exhibit a great fluctuation in write requests.

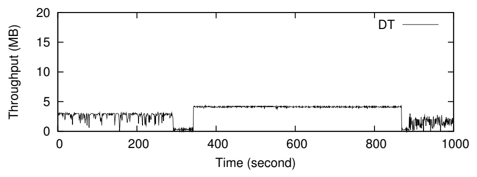
5 Related Work

There have been a lot of studies on improving the endurance of flash-based SSDs. Many existing garbage collection and wear-leveling techniques [27, 28, 29, 30, 31, 32, 33, 34] are designed to improve the lifetime of SSDs by avoiding useless data migration during a block recycling process or by distributing P/E cycles of flash blocks as evenly as possible.

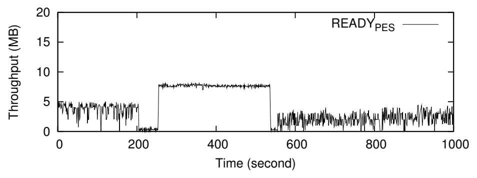
As the endurance of flash memory continuously dete-



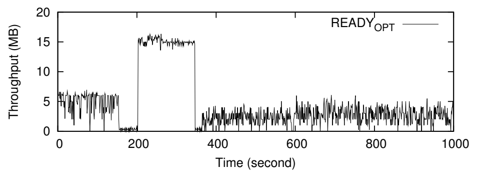
(a) ST



(b) DT



(c) READY$_{\mathrm{PES}}$



(d) READY$_{\mathrm{OPT}}$

Figure 14: A detailed analysis of four SSD configurations with the `map` trace.

riorates, several endurance enhancement techniques that aggressively reduce the number of data written to SSDs have been proposed. Data de-duplication [26, 35, 36] and data compression [37, 38] are representative examples of these policies. Data de-duplication detects duplicate data blocks that already exist in a storage device and then eliminates redundant writes to SSDs for such blocks. Data compression eliminates repeated bit patterns within a data block, reducing writes to SSDs. These techniques are useful in improving the lifetime of SSDs, but they have some limitations in that none of them guarantee the SSD lifetime or make use of the recovery effect of a memory cell.

More recently, the approaches that exploit the recovery effect of flash devices have received considerable attention. This paper is an improved version of our preliminary work [39]. Mohan *et al.* investigated the effect of the damage recovery on the lifetime of SSDs for enterprise servers [1]. They claimed that the endurance of NAND flash memory was durable enough even for I/O intensive enterprise applications because of its recovery ability. However, their investigations were limited to 90 nm SLC and MLC flash memories which exhibit good endurance properties. They also did not exploit the benefit of the recovery effect in ensuring the lifetime of SSDs. Wu *et al.* presented an endurance enhancement technique that boosts recovery speed by heating a flash chip worn out under high temperature [10]. By leveraging the temperature-accelerated recovery, it improved the endurance of SSDs up to five times. However, one of the major drawbacks of this approach is that it requires extra energy consumption to heat flash chips, lowering the energy efficiency of a storage device. Unlike Wu's work, our study considers the endurance improvement of SSDs at the room temperature and exploits this benefit to guarantee the lifetime of SSDs with less throttling overhead.

6 Conclusions

In this paper, we proposed a recovery-aware dynamic throttling technique, READY, to overcome two main problems in realizing the adoption of SSDs in enterprise server systems: the continuously decreasing endurance and unpredictable lifetime problems. READY throttles write performance so that the required lifetime of SSDs is to be satisfied. In order to guarantee the SSD lifetime with less throttling overhead, READY exploits the recovery effect of a floating-gate transistor which effectively increases the number of effective P/E cycles of SSDs. Our evaluation results showed that the proposed throttling technique guarantees a lifetime warranty, while achieving a relatively small reduction in write response time and little response time variation over the static throttling technique.

READY can be improved in several directions. The stress and recovery model of this work is based on the previous studies on the physical characteristics of flash memory [1, 2, 10]. These studies carefully modeled the stress and recovery characteristics of flash memory, but their scopes were limited to NOR or NAND flash memory fabricated in over 90 nm technology. To build a more accurate stress and recovery model, we will perform investigations using real NAND flash parts which are fabricated in less than 30 nm technology. We also plan to implement READY in a real SSD platform to evaluate its effectiveness in real systems.

Acknowledgments

We would like to thank Daniel Peek, our shepherd, and anonymous referees for valuable suggestions that greatly improved the paper. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 20110020426, No. R33-10095, and No. 2011-0020514). Samsung Electronics partially supported our research and the ICT at SNU provided research facilities for this study.

References

[1] V. Mohan, T. Siddiqua, S. Gurumurthi, and M. Stan, "How I Learned to Stop Worrying and Love Flash Endurance," in *Proceedings of the Workshop on Hot Topics in Storage and File Systems*, 2010.

[2] N. Mielke, H. Belgal, A. Fazio, Q. Meng, and N. Righos, "Recovery Effects in the Distributed Cycling of Flash Memories," in *Proceedings of the IEEE International Reliability Physics Symposium*, 2006.

[3] B. You and et. al, "A High Performance Co-design of 26 nm 64 Gb MLC NAND Flash Memory using the Dedicated NAND Flash Controller," *Journal of Semiconductor Technology and Science*, vol. 11, no. 2, pp. 121-129, 2011.

[4] N. Sommer, "Signal Processing and the Evolution of NAND Flash Memory," *Embedded Computing Design*, vol. 8, no. 8, 2010.

[5] Y. Koh, "NAND Flash Memory Scaling Beyond 20 nm," in *Proceedings of the IEEE International Memory Workshop*, 2009.

[6] M. Goldman, K. Pangal, G. Naso, and A. Goda, "25nm 64Gb 130mm^2 3bpc NAND Flash Memory," in *Proceedings of the International Memory Workshop*, 2011.

[7] Micron Technology Inc., "An Enterprise-Focused MLC SSD," `http://www.micron.com/products/solid_state_storage/enterprise_ssd/p400e.html`, 2011.

[8] SandForce Inc., "SF-2500 & SF-2600 Enterprise SSD Processors," `http://www.sandforce.com/index.php?id=21&parentId=2`, 2010.

[9] C. Black, "24 Months of Intel SSDs... What We've Learned about MLC in the Enterprise...," *Intel Open Port IT Community*, 2011.

[10] Q. Wu, G. Dong, and T. Zhang, "Exploiting Heat-Accelerated Flash Memory Wear-Out Recovery to Enable Self-Healing SSDs," in *Proceedings of the Workshop on Hot Topics in Storage and File Systems*, 2011.

[11] R. Yamada, T. Sekiguchi, Y. Okuyama, J. Yugami, and H. Kume, "A Novel Analysis Method of Threshold Voltage Shift due to Detrap in a Multi-Level Flash Memory," in *Proceedings of the Symposium on VLSI Technology*, 2001.

[12] H. Yang and et. al, "Reliability Issues and Models of Sub-90nm NAND Flash Memory Cells," in *Proceedings of the International Conference on Solid-State and Integrated Circuit Technology*, 2006.

[13] Y. Pan, G. Dong, and T. Zhang, "Exploiting Memory Device Wear-Out Dynamics to Improve NAND Flash Memory System Performance," in *Proceedings of the USENIX Conference on File and Storage Technologies*, 2011.

[14] S. Gurumurthi, A. Sivasubramaniam, and V. Natarajan, "Disk Drive Roadmap from the Thermal Perspective: A Case for Dynamic Thermal Management," in *Proceedings of the International Symposium on Computer Architecture*, 2005.

[15] L. Grupp, A. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. Siegel, and J. Wolf, "Characterizing Flash Memory: Anomalies, Observations, and Applications," in *Proceedings of the International Symposium on Microarchitecture*, 2009.

[16] S. Boboila and P. Desnoyers, "Write Endurance in Flash Drives: Measurements and Analysis," in *Proceedings of the USENIX conference on File and Storage Technologies*, 2010.

[17] SanDisk, "Longterm Data Endurance (LDE) for Client SSD," `http://www.sandisk.com/Assets/File/pdf/oem/LDE/WhitePaper.pdf`, 2008.

[18] Silicon Graphics International, "Solid State Disk Solutions," `http://www.sgi.com/products/storage/ssd/endurance.html`.

[19] SMART Modular Technologies, "XceedIOPS SATA SSD," `http://www.smartm.com/products/storage/SSDs.asp`.

[20] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper, "Workload Analysis and Demand Prediction of Enterprise Data Center Applications," in *Proceedings of the IEEE International Symposium on Workload Characterization*, 2007.

[21] D. Narayanan, A. Donnelly, and A. Rowstron, "Write Off-Loading: Practical Power Management for Enterprise Storage," in *Proceedings of the USENIX Conference on File and Storage Technologies*, 2008.

[22] S. Kavalanekar, B. Worthington, Q. Zhang, and V. Sharda, "Characterization of Storage Workload Traces from Production Windows Servers," in *Proceedings of the International Symposium on Workload Characterization*, 2008.

[23] N. Agrawal, V. Prabhakaran, and T. Wobber, "Design Tradeoffs for SSD Performance," in *Proceedings of the USENIX Annual Technical Conference*, 2008.

[24] X. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write Amplification Analysis in Flash-based Solid State Drives," in *Proceedings of the Israeli Experimental Systems Conference*, 2009.

[25] Fusion-io Inc., "MySpace Uses Fusion Powered I/O to Drive Greener and Better Data Centers," `http://www.fusionio.com/case-studies/myspace-case-study.pdf`, 2010.

[26] Q. Yang and J. Ren, "I-CASH: Intelligently Coupled Array of SSD and HDD," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2011.

[27] J. Kim, J.-M. Kim, S.-H. Noh, S.-L. 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.

[28] S.-W. Lee, D.-J. Park, T.-S. Chung, W.-K. Choi, D.-H. Lee, S.-W. Park, and H.-J. Song, "A Log Buffer Based Flash Translation Layer Using Fully Associative Sector Translation," *ACM Transactions on Embedded Computing Systems*, vol. 6, no. 3, 2007.

[29] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee, "A Superblock-Based Flash Translation Layer for NAND Flash Memory," in *Proceedings of the International Conference on Embedded Software*, 2006.

[30] 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.

[31] H. Kim and S. Lee, "A New Flash Memory Management for Flash Storage System," in *Proceedings of the Computer Software and Applications Conference*, 1999.

[32] M.-L. Chiang, P.-H. Lee, and R.-C. Chang, "Cleaning Policies in Mobile Computers using Flash Memory," *Journal of Systems and Software*, 1999.

[33] L.-P. Chang, "On Efficient Wear Leveling for Large-Scale Flash-Memory Storage Systems," in *Proceedings of the ACM Symposium on Applied Computing*, 2007.

[34] D. Jung, Y.-H. Chae, H. Jo, J.-S. Kim, and J. Lee, "A Group-Based Wear-Leveling Algorithm for Large-Capacity Flash Memory Storage Systems," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2007.

[35] 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 USENIX Conference on File and Storage Technologies*, 2011.

[36] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam, "Leveraging Value Locality in Optimizing NAND Flash-Based SSDs," in *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, 2011.

[37] K. Yim, H. Bahn, and K. Koh, "A Flash Compression Layer for Smartmedia Card Systems," *IEEE Transactions on Consumer Electronics*, 2004.

[38] T. Park and J.-S. Kim, "Compression Support for Flash Translation Layer," in *Proceedings of the International Workshop on Software Support for Portable Storage*, 2010.

[39] S. Lee, T. Kim, K. Kim, and J. Kim, "Guaranteeing the Lifetime of SSDs Using Recovery-Aware Dynamic Throttling," in *Proceedings of the International Memory Architecture and Organization Workshop*, 2011.