conference

proceedings

FAST '10: 8th USENIX Conference on File and Storage Technologies

San Jose, CA, USA
February 23–26, 2010

Sponsored by

USENIX

in cooperation with
ACM SIGOPS

USENIX

Proceedings of FAST '10: 8th USENIX Conference on File and Storage Technologies

San Jose, CA, USA February 23–26, 2010

ISBN 978-1-931971-74-4

USENIX Association

Proceedings of FAST '10:
8th USENIX Conference on File and Storage Technologies

February 23–26, 2010
San Jose, CA, USA

Conference Organizers

Program Co-Chairs

Randal Burns, *Johns Hopkins University*
Kimberly Keeton, *Hewlett-Packard Labs*

Program Committee

Patrick Eaton, *EMC*
Jason Flinn, *University of Michigan*
Gary Grider, *Los Alamos National Lab*
Ajay Gulati, *VMware*
Sudhanva Gurumurthi, *University of Virginia*
Dushyanth Narayanan, *Microsoft Research*
Jason Nieh, *Columbia University*
Christopher Olston, *Yahoo! Research*
Hugo Patterson, *Data Domain*
Beth Plale, *Indiana University*
James Plank, *University of Tennessee*
Erik Riedel, *EMC*
Alma Riska, *Seagate*
Steve Schlosser, *Avere Systems*
Bianca Schroeder, *University of Toronto*
Karsten Schwan, *Georgia Institute of Technology*
Craig Soules, *Hewlett-Packard Labs*
Alan Sussman, *University of Maryland*
Kaladhar Voruganti, *NetApp*
Hakim Weatherspoon, *Cornell University*
Brent Welch, *Panasas*
Ric Wheeler, *Red Hat*
Yuanyuan Zhou, *University of California, San Diego*

Work-in-Progress Reports (WiPs) and Poster Session Chair

Hakim Weatherspoon, *Cornell University*

Tutorial Chair

David Pease, *IBM Almaden Research Center*

Steeering Committee

Andrea C. Arpaci-Dusseau, *University of Wisconsin—Madison*
Remzi H. Arpaci-Dusseau, *University of Wisconsin—Madison*
Mary Baker, *Hewlett-Packard Labs*
Greg Ganger, *Carnegie Mellon University*
Garth Gibson, *Carnegie Mellon University and Panasas*
Peter Honeyman, *CITI, University of Michigan, Ann Arbor*
Darrell Long, *University of California, Santa Cruz*
Jai Menon, *IBM Research*
Erik Riedel, *EMC*
Margo Seltzer, *Harvard School of Engineering and Applied Sciences*
Chandu Thekkath, *Microsoft Research*
Ric Wheeler, *Red Hat*
John Wilkes, *Google*
Ellie Young, *USENIX Association*

The USENIX Association Staff

External Reviewers

Nitin Agrawal
Lakshmi Bairavasundaram
Alexandros Batsakis
John Bent
Emery Berger
Luc Bouganim
Jeff Butler
Lucy Cherkasova
Fred Douglis
David Du
Daniel Ellard
Khaled Elmeleegy
Michael Factor
Kevin Greenan
John Linwood Griffin
Jim Hafner
Stavros Harizopoulos
Ragib Hasan
Paul Jardetzky
Song Jiang
Arkady Kanevsky
Christos Karamanolis
Andy Klosterman
Ed Lee
Xiaoizhou Li
Mark Lillibridge
Chris Lumb
Pramod Mandagere
Jeanna Matthews
Vipul Mathur
Arif Merchant
Mike Mesnier
Ethan Miller
Brad Morrey
Kiran-Kumar Muniswamy-Reddy
David Pease
Zachary Peterson
Eduardo Pinheiro
Raju Rangaswami
Ben Reed
Brandon Salmon
Jiri Schindler
Mehul Shah
Keith Smith
Kiran Srinivasan
Deepti Srivastava
Sai Susarla
Renu Tewari
Douglas Thain
Eno Thereska
Bhuvan Urgaonkar
Mustafa Uysal
Elizabeth Varki
Neal Walfield
Jun Wang
Sage Weil
Theodore Wong
Jay Wylie

FAST '10: 8th USENIX Conference on File and Storage Technologies
February 23–26, 2010
San Jose, CA, USA

Wednesday, February 24

Build a Better File System and the World Will Beat A Path to Your Door.

Looking for Trouble

Thursday, February 25

Flash: Savior of the Universe?

Thursday, February 25 (continued)

I/O, I/O, to Parallel I/O We Go

Making Management More Manageable

Friday, February 26

Concentration: The Deduplication Game

The Power Button

Message from the Program Co-Chairs

Dear Colleagues,

We welcome you to the 8th USENIX Conference on File and Storage Technologies (FAST '10). This year we are proud to carry on the FAST tradition of presenting high-quality, innovative file and storage systems research. The program includes papers on emerging hot topics, with contributions to solid-state storage technology, power-efficient storage systems, and dealing with latent errors. It displays the breadth of storage systems research with sessions on parallel I/O and deduplication. It also contains significant contributions to the core of the field with sessions on storage management and file systems.

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, industrial labs, national labs, and collaborations thereof.

FAST '10 received 89 submissions, from which 18 papers were selected, for an acceptance rate of 20%. Each paper received at least three reviews from PC members. All but two papers received four or more reviews. The 371 total reviews consist of 295 PC reviews and 76 reviews from 58 external reviewers.

The review process was conducted online over two months and at a program committee meeting held in Palo Alto, CA, in early November 2009. We used Eddie Kohler's HotCRP software to handle paper submissions, reviews, PC discussion, and notifications. Initially, reviews for each paper were assigned to four PC members or to three PC members and an external reviewer. Then, controversial papers—those with both strong negative and positive reviews—were discussed online and additional reviews were obtained for many such papers. 20 of the 23 PC members attended the PC meeting, at which the program was selected, in person. In addition to technical merit, the discussion at the PC meeting focused on whether papers were new and exciting, of broad interest to the FAST community, and likely to generate controversy and discussion. These factors weighed heavily in paper selection.

It was an absolute pleasure to assemble this program, and we would like to thank everyone who contributed. First and foremost, we are indebted to all of the authors who submitted papers to FAST '10. 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 '10 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 dealt with innumerable issues large and small and provided outstanding technical and emotional support. They are pleasant and professional and 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. Serving on the FAST PC involves lots of reading, writing many lengthy reviews, participating in online discussion, and traveling. FAST and other USENIX systems conferences are distinguished by continuing to have in-person PC meetings. The discussion that happened at the PC meeting was invaluable in identifying the most exciting papers to include in the program.

We look forward to seeing you in San Jose!

Randal Burns, ***Johns Hopkins University***
Kimberly Keeton, ***Hewlett-Packard Labs***
Program Co-Chairs

quFiles: The right file at the right time

Kaushik Veeraraghavan[*]*, Jason Flinn*[*]*, Edmund B. Nightingale*[†] *and Brian Noble*[*]
University of Michigan[*] *Microsoft Research (Redmond)*[†]

Abstract

A quFile is a unifying abstraction that simplifies data management by encapsulating different physical representations of the same logical data. Similar to a quBit (quantum bit), the particular representation of the logical data displayed by a quFile is not determined until the moment it is needed. The representation returned by a quFile is specified by a data-specific policy that can take into account context such as the application requesting the data, the device on which data is accessed, screen size, and battery status. We demonstrate the generality of the quFile abstraction by using it to implement six case studies: resource management, copy-on-write versioning, data redaction, resource-aware directories, application-aware adaptation, and platform-specific encoding. Most quFile policies were expressed using less than one hundred lines of code. Our experimental results show that, with caching and other performance optimizations, quFiles add less than 1% overhead to application-level file system benchmarks.

1 Introduction

It has become increasingly common for new storage systems to implement *context-aware adaptation*, in which different representations of the same object are returned based on the context in which the object is accessed. For instance, many systems transcode data to meet the screen size constraints of mobile devices [5, 12]. Others display reduced fidelity representations to meet constraints on resources such as network bandwidth [8, 27] and battery energy [11], display redacted representations of data files when they are viewed at insecure locations [22, 42], and create different formats of multimedia data for diverse devices [29].

These systems, and many others, have been successful at addressing specific needs for adapting the representation of data to fit a given context. However, they suffer from several problems that inhibit their wide-scale adoption. First, building such systems is time-consuming. Most required several person-years to build a prototype; porting them to mainstream environments would be difficult at best. Second, each system presents a different abstraction and interface, so each has a learning curve. Third, these systems typically present only a single logical view of data, making it difficult for users to pierce the abstraction and explicitly choose different presentations.

Why are there so many systems that share the same premise, yet have completely separate implementations? The answer is that, as a community, we have failed to recognize that there is a fundamental abstraction that underlies all these systems. This simple abstraction is the ability to view different representations of the same logical data in different contexts.

In this paper, we argue that this new abstraction, which we refer to as a quFile, should be implemented as a first-class file system entity. A quFile encapsulates different physical representations of the same logical data. Similar to a quBit (quantum bit), the particular representation of the logical data displayed by a quFile is not determined until the moment it is needed. The representation returned by a quFile is specified by a data-specific policy that can take into account context such as the application requesting the data, the device on which data is accessed, screen size, and battery status.

quFiles provide a *mechanism/policy split*. In other words, they provide a common mechanism for dynamically resolving logical data items to specific representations in different contexts. A common mechanism reduces the time to develop new context-sensitive systems; developers only need to write code that expresses their new policies because quFiles already provide the mechanism. A common mechanism also makes deploying new systems easier. Since the file system provides a unifying mechanism, a new policy can be inserted simply by creating another quFile.

quFiles provide *transparency* for quFile-unaware users and applications. Each quFile policy defines a *default view* that makes the observable behavior of the file system indistinguishable from the behavior of a file system without quFiles that happens to contain the correct data for the current context. This transparency has a powerful property: no application modification is required to benefit from quFiles. The default view also provides *encapsulation* by hiding the messy details of the physical representation and exporting only a context-specific logical view of the data.

For users and applications that are quFile-aware, a single logical representation of the data is often not enough. For instance, some users may wish to view the data in the quFile as it is actually stored or see a different logical presentation of data than the one provided by default. quFiles support this functionality through their

views interface. All quFiles export a *raw view* that allows the physical representation of data within a quFile to be directly viewed and manipulated. In addition, quFile policies may define any number of *custom views*, each of which is an alternate logical representation of the data contained within the quFile. Users and applications select views using a special filename suffix, an interface that allows users to select views even when using unmodified commercial-off-the-shelf (COTS) applications.

How good is the quFile abstraction? We demonstrate its generality by implementing both ideas previously proposed by the research community (application-aware adaptation, copy-on-write file systems, location-aware document redaction, and platform-specific caching) and new ideas enabled by the abstraction (using spare storage to save battery energy and resource-aware directories). Our experience suggests a "natural fitness" for implementing context-aware policies using quFiles: compared to the multiple developer-years required to implement each of the existing systems described above, a single graduate student implemented each new policy in less than two weeks using quFiles. Further, policies required only 84 lines of code on average. Our results show that, with caching and other performance optimizations, quFiles add less than 1% overhead to application-level file system benchmarks.

2 Related Work

A quFile is a new abstraction that encapsulates different physical representations of the same logical data and dynamically returns the correct representation of the logical data for the context in which it is accessed.

quFiles are not an extensibility mechanism. Instead, they are an abstraction that uses safe extensibility mechanisms (Sprockets [30] in our implementation) to execute policies. Thus, quFiles could use previously-proposed operating system extensibility mechanisms such as Spin [3], Exokernel [10], or Vino [39], as well as file system extensibility mechanisms such as Watchdogs [4] or FUSE [13]. Compared to Watchdogs and FUSE, quFiles present a minimal interface that focuses on contextual awareness; this results in policies that can be expressed in only a few lines of code.

A quFile can be thought of as the file system equivalent of a materialized view in a relational database [17]. Unlike materialized views, quFiles return different data depending on the context in which they are accessed, and they operate on file data, which has no fixed schema. Similarly, OdeFS [14] presents a transparent file system view of data stored in a relational database. However, unlike quFiles, OdeFS objects are always statically resolved to the same view.

Multiple systems adapt the fidelity of data presented to clients. Since a full discussion of this body of work is outside the scope of this paper, we only list here those systems that directly inspired our quFile case studies. These include systems that transcode data to meet screen size constraints [12], network bandwidth limitations [8, 27], battery energy constraints [11], format decoding limitations [29], or storage restrictions [33]. These previous systems either require application or operating system modification or the addition of an intermediary proxy that performs data adaptation. With quFiles, we propose a unified mechanism within the file system that can dynamically invoke any adaptation policy.

To simplify data management across multiple devices, Cimbiosys [34], PRACTI [2], and Perspective [36] allow clients to specify which files to replicate with query-based filters. quFiles could complement filters by adding context-awareness to replication policies.

Some file systems allow limited dynamic resolution of file content. Mac OS X Bundles [6] are file system directories that resolve to a platform-specific binary when accessed through the Mac OS X Finder. Similarly, AFS [18] has an "@sys" directory that resolves to the binary appropriate for a particular client's architecture. quFiles are a more general abstraction that capture these specific instances that embed particular resolution policies into the file system. NTFS has Alternate Data Streams [35] that support multiple representations of data within a file. However, unlike quFiles, NTFS does not currently support safe execution of arbitrary application policies to determine which representation should be accessed.

We describe one metadata edit policy for low-fidelity files. Other quFile policies could be implemented to support adaptation-aware editing [7]. One possible approach is to layer updates separately from the data they modify and reconcile the high-fidelity original with the edit layer at a later time [32].

Past approaches such as Xerox's Placeless Documents [9] and Gifford's Semantic File Systems [15] suggest semantic or property-based mechanisms to better organize and manage data in a file system. quFiles share the same goals of improving organization and simplifying management, but we have chosen a backward-compatible design that works within existing file systems, rather than requiring a system re-write. The Semantic File System provides virtualized directories of files with similar attributes, whereas quFiles virtualize name and content of data within a directory based on context.

Schilit et al. advocate context-aware computing applications [38] and identify four major categories of applications. Of these, quFiles support context-triggered actions, as well as contextual information and command-based applications. While Schilit et al. focus on usability and the graphical user interface, quFiles focus on supporting different views of the data in the file system.

Building on these ideas, context-aware middleware [21] allows applications to modify the presentation of data depending on access context. However, these systems require application modification, e.g., to subscribe to context events. quFiles provide similar functionality transparently to unmodified applications by manipulating the file system interface.

3 Design goals

We next describe the goals that we aimed to achieve with our design of quFiles.

3.1 Be transparent to the quFile-unaware

We designed quFiles to be transparent by default. quFiles hide their presence from users and applications unaware of their existence. We say quFiles are transparent if *the observable behavior of a file system containing quFiles is indistinguishable from the behavior of a file system without quFiles that contains the correct data for the current context.* Consider a quFile that contains multiple formats of a video and returns the one appropriate for the media player that accesses the data. In this case, the application need not be aware of the quFile. It perceives that the file system contains a single instance of the video that happens to be one it can play. In general, a quFile may dynamically resolve to zero, one, or many files located in the directory in which it resides; we refer to this logical representation as the quFile's *default view*.

The default view provides the backward compatibility required to use COTS applications. Without modification, such applications must be quFile-unaware, so the context-specific presentation of data must be accomplished by presenting the illusion of a file system without quFiles that contains the appropriate data. The default view also reduces the cognitive load on the user by removing the need to reason about which representation of data should be accessed in the current context. Instead, the policy executed by the quFile mechanism makes this decision transparently.

Note that our definition of transparency applies to any specific point in time. When context changes, the appropriate representation to return may also change. This implies that a quFile-unaware user or application may observe that the contents of the file system change over time. This behavior is the same as that seen when another application or user modifies a file. For instance, a quFile may redact files to remove sensitive content when data is accessed at insecure locations. A user will necessarily notice that the contents of the file change after moving from home to a coffee shop. However, the quFile *mechanism* itself remains transparent, so the same application can display the file in both contexts.

3.2 Don't hide power from the quFile-aware

A quFile does not hide power from users and applications that wish to view and manipulate data directly. Instead, a quFile allows them to select among different *views*, each of which is a different presentation of its data. In addition to the default view described in the previous section, each quFile also presents a *raw view* that shows the data within the quFile as files within a directory. The raw view might include, for example, an original object, all materialized alternate representations of that object, as well as the links to policies that govern the quFile. quFile-aware utilities typically use the raw view to manipulate quFile contents directly.

The raw and default views represent the two endpoints on the spectrum of transparency. In between, a quFile's policy may define any number of additional *custom views*. A custom view returns a different logical representation of the data than that provided by the default view. A quFile-aware user or application can specify the name of a custom view when accessing a quFile to switch to an alternate representation. In effect, the name of the custom view becomes an additional source of context.

For example, consider a quFile that keeps old versions of a file for archival purposes along with the file's current version. The quFile's default view returns a representation equivalent to the file's current version. In the common case, the file system is as easy to use as one that does not support versioning because its outward appearance is equivalent to that of one without versioning. However, when a backup version is needed, the user should be able to see all the previous versions of the file and select the correct representation. The quFile policy therefore defines a `versions` custom view that shows all past versions in addition to the current one. Another custom view (a `yesterday` view) might show the state of all files as they existed at midnight of the previous day, and so on. Finally, a utility that removes older versions to save disk space may need to see incremental change logs, not just checkpoints, so that it can compact delta changes to reduce storage use. This utility uses the quFile's raw view.

quFiles distinguish between application transparency and user transparency. In the above example, a user may view previous versions of a file using `ls` or a graphical file browser. The user is quFile-aware, but the file browser is quFile-unaware. This scenario is tricky because the user must pass quFile-specific information through the unmodified application to the quFile policy. We solve this dilemma by using the file name, which is generally treated as a black box by applications to encode view selection. Specifically, for a directory `papers`, the user may select the versions custom view by specifying the name `papers.quFile.versions` or the raw view by specifying `papers.quFile`, which is shorthand for `papers.quFile.raw`.

3.3 Support both static and dynamic content

quFiles support both static and dynamic content. When data is read from a quFile, the file names and content returned might either be that of files stored within the quFile or new values generated on the fly. Storing and returning static content within the quFile amortizes the work of generating content across multiple reads. Static content can also reduce the load on resource-impoverished mobile devices; e.g., rather than transcode a video on demand on a mobile computer, we pre-transcode the video on a desktop and store the result in a quFile. On the other hand, dynamic content generation is useful when all context-dependent versions cannot be enumerated easily. For instance, our versioning quFile dynamically creates checkpoints of files at specific points in time from an undo log of delta changes.

3.4 Be flexible for policy writers

quFiles support not just the resolution policies that we have implemented so far, but also resolution policies that we have yet to imagine. We provide this flexibility by allowing resolution policies to be specified as short code modules in libraries that are dynamically loaded when a quFile is accessed. Each quFile links to the specific policies that govern it: a `name` policy that determines its name(s) in a given context, a `content` policy that determines its contents in a given context, and an `edit` policy that describes how its contents may be modified. A quFile may optionally link to two `cache` policies that direct how its contents are cached. These policies are easy to craft; the policies for our six case studies average only 84 lines of code.

Executing arbitrary code within the file system is dangerous, so policies are executed in a user-level sandbox. Our current implementation can use Sprocket [30] software fault isolation to ensure that buggy policies do not damage the file system or consume unbounded resources (e.g., by executing an infinite loop); other safe execution methods should work equally well.

4 Implementation

4.1 Overview

To illustrate how quFiles work, we briefly describe one quFile we developed. This quFile returns videos formatted appropriately for the device on which the video is viewed. When a new video is added to the file system, a quFile-aware transcoder utility learns of the new file through a file system change notification. The transcoder creates alternate representations of the video sized and formatted for display on the different clients of the file system. It then creates a quFile and moves the original and alternate representations into the quFile using the quFile's raw view.

The transcoder also sets specific policies that govern the behavior and resolution of the quFile. A `name` policy determines the name of a quFile in a given context. If the quFile dynamically resolves to multiple files, the policy returns all resolved names in a list. For example, one author owns a DVR that displays only TiVo files, which must have a file name ending in `.TiVo`. The `name` policy thus returns `foo.TiVo` when a video is viewed using the DVR and `foo.mp4` otherwise.

A `content` policy determines the content of the quFile in a given context. This policy is called once for each name returned by a quFile's `name` policy. In the video example, the `content` policy returns the alternate representation in the TiVo format when the quFile is viewed on the DVR, an alternate representation for a smaller screen size when the quFile is viewed on a Nokia N800 Internet tablet, and the original representation when the quFile is viewed on a laptop. Note that the example quFile resolves to the same name on the N800 and the laptop, yet it resolves to different content on each device. Thus, COTS video players see only the video in the format they can play. Users who are quFile-unaware see the same video when they list the directory, but a quFile-aware power user could use the raw view to see all transcodings.

An `edit` policy specifies whether specific changes are allowed to the contents of a quFile. For instance, the user may modify the metadata of a lower-fidelity representation on the N800. In this case, the video transcoder is notified of the edit, and it makes corresponding modifications to the metadata of the other representations. However, changes to the actual video are disallowed since there is no easy way to reflect changes made to a low-fidelity version to higher-fidelity representations.

Two optional `cache` policies specify context-aware prefetching and cache eviction policies for the quFile and its contents. These policies help manage the cache of distributed file systems [18, 20, 26] that persistently store data on the disk of a file system client. For the example quFile, the `cache` policies ensure that only the format needed for a specific device is cached on that device.

4.2 Background: BlueFS

The quFile design is sufficiently generic so that quFile support can be added to most local and distributed file systems. For our prototype implementation, we added quFile support to the Blue File System [26] (BlueFS) because BlueFS targets mobile and consumer usage scenarios for which quFiles are particularly useful and because we were familiar with the code base. BlueFS is an open-source, server-based distributed file system with support for both traditional computers and mobile devices such as cell phones. Additionally, BlueFS can cache data on a device's local storage and on removable storage media to improve performance and support disconnected operation [20]. BlueFS has a small kernel module that man-

```
name_policy             (IN list of quFile contents, IN view name (if specified),
                         OUT list of file names, OUT cache lifetime);
content_policy          (IN filename, IN list of quFile contents, IN view name (if specified),
                         OUT fileid, OUT cache lifetime);
edit_policy             (IN fileid, IN edit type, IN offset, IN size, OUT enum {ALLOW, DISALLOW, VERSION})
cache_insert_policy     (IN list of quFile contents, OUT list of fileids to cache)
cache_eviction_policy   (IN fileid, OUT enum {EVICT, RETAIN})
```

Figure 1. quFile API

ages file system data in the kernel's caches. The kernel module redirects most VFS operations to a user-level daemon. To support quFiles, we made small modifications to both the kernel module and daemon, while the file server remained unchanged. For simplicity, we also use BlueFS' persistent query [29] mechanism to deliver file change notifications.

4.3 Physical representation of a quFile

Logically, a quFile is a new type of file system object. A quFile is similar to a directory in that they both contain other file system objects. The difference between quFiles and directories is their resolution policies. Directory resolution policies are *static*: given the same content, a directory returns the same results. quFile resolution policies are *dynamic*: the same content may resolve differently in different contexts. Further, users and applications must be aware of directories since they add another layer to the file system hierarchy, whereas quFiles can hide their presence by simply adding resolved files to the listing of their parent directories.

Using this observation, we reduce the amount of new code required to add quFiles to a file system by having the physical (on-disk and in-memory) representation of a quFile be the same as a directory, but we redefine a quFile's VFS operations to provide different functionality than that provided by a directory. We segment the namespace to differentiate quFiles from regular directories. All quFiles have names of the form <name>`.quFile`. While we considered other methods of differentiating the two, such as using a different file mode, a special filename extension allows quFile-aware utilities to manipulate quFiles without changing the file system interface. For example, the video transcoder simply issues the commands `mkdir foo.quFile` and `mv /tmp/foo.mp4 foo.quFile` to create a quFile and populate it with the original video. The only disadvantage of namespace differentiation is the unlikely possibility that a quFile-unaware application might try to create a directory that ends with `.quFile`. Note that the quFile-aware transcoder uses the quFile's raw view to manipulate its contents; this allows it to use COTS file system utilities such as `mv`. Video players will see the default view since they will not use the special `.quFile` extension. When they list the directory containing the quFile, they will see an entry for either `foo.mp4` or `foo.TiVo`.

4.4 quFile policies

Figure 1 shows the programming interface for all quFile policies. Policies are stored in shared libraries in the file system. When a quFile is created, utilities such as the video transcoder create links in the quFile to the libraries for its specific policies. Links share policies across quFiles of the same type, simplifying management and reducing storage usage.

4.4.1 Name policies

A `name` policy lets a quFile have different logical names in different contexts. To make the existence of a quFile transparent to quFile-unaware applications and users, a VFS `readdir` on the parent directory of a quFile does not return the quFile's name; instead, it returns the names of zero to many logical representations of the data encapsulated within the quFile. quFiles interpose on the parent's `readdir` because that is when the filenames of the children of a directory are returned to an application.

If `readdir` encounters a directory entry with the reserved `.quFile` extension, it makes a downcall to the BlueFS daemon, which runs the `name` policy for that quFile. The kernel reads the quFile's static contents from the page cache and passes the contents to the daemon.

The user may optionally specify the name of a view for the `name` policy. For example, instead of typing `ls foo`, a user could type `ls foo.quFile.versions` to show a directory listing that contains all versions retained by the quFiles in the directory. The view name is passed to the `name` policy without interpretation by the file system. This allows a quFile-aware user to use a COTS application such as `ls` to list file versions when desired. As mentioned previously, the syntax `ls foo.quFile` returns the raw view of the quFile, which shows the quFile and all its contents as a subdirectory within `foo`. This syntax allows quFile-aware utilities and users to directly manipulate quFile contents and policies.

The `name` policy returns a list of zero to many logical names. The kernel module then calls `filldir` for each name on the list to return them to the application reading the directory. If no names are returned by the policy, the kernel does not call `filldir`. This hides the existence of the quFile from the application.

In addition to returning the name of existing representations encapsulated in a quFile, a `name` policy may also dynamically instantiate new representations by returning filenames that do not currently exist within the

quFile. To ensure that such names do not conflict with other directory entries or names returned by other quFiles within the directory, each quFile reserves a portion of the directory namespace. For instance, the names returned by `foo.quFile` must all start with the string `foo`; e.g., `foo.mp3`, `foo.bar.txt`, etc. Directory manipulation functions such as `create` and `rename` ensure that the claimed namespace does not conflict with current directory entries. For example, creating a quFile `foo.quFile` is disallowed if there currently exists within the directory a file named `foo.txt` or another quFile named `foo.tex.quFile`.

To improve performance, a `name` policy may specify a cache lifetime for the names it returns — the kernel will not re-invoke the `name` policy for this time period. By default, the kernel module does not cache entries if no lifetime is specified, so the policy is reinvoked on the next `readdir` and may return different entries if context has changed. Cache lifetimes are useful for policies that depend on slowly-changing context such as battery life.

4.4.2 Content policies

A `content` policy lets a quFile have different content in different contexts. After reading a directory, an application that is unaware of quFiles will believe that there are one or more files with the logical names returned by the quFile's `name` policy within that directory. Thus, it issues a VFS `lookup` for each logical name. Since no such file exists, we modify `lookup` to return an inode of a file containing the logical content associated with the name in the given context.

The modified BlueFS `lookup` operation checks whether the name being looked up resides within the directory namespace reserved by a quFile. If this is the case, it makes a downcall to the BlueFS daemon, passing the filename being looked up, a list of the quFile's contents, and a view name if one was specified. The daemon calls the quFile's `content` policy, which returns the unique identifier of a file containing the appropriate content. The kernel module `lookup` operation instantiates a Linux `dentry` with the inode specified by the fileid returned by the policy.

This implementation allows quFiles to create content dynamically. A `content` policy can first create a new file and populate it with content, then return the newly created file to the kernel. Like `name` policies, `content` policies may also specify a cache lifetime for the content they return. If a lifetime is not specified, the kernel does not cache the resulting `dentry`, which forces a new `lookup` the next time the content is accessed.

4.4.3 Edit policies

An `edit` policy specifies which modifications to a quFile's contents are allowed. Currently, quFiles support three actions: the modification can be allowed, disallowed, or force the creation of a new version. We modified VFS operations such as `commit_write` and `unlink` to make a downcall to the daemon when a quFile representation is modified. The daemon runs the `edit` policy, passing in the unique identifier of the file being modified and the type of the modifying operation. For write operations, it also specifies the region of the file being modified. The policy returns an enum that specifies which action to take.

If the edit is allowed, the modification proceeds as normal. If it is disallowed, the kernel returns an error code to the calling application specifying that the file is read-only. If the edit should cause a new version, we modify the representation in place but also save the previous version of the modified range in an undo log. We chose to log changes rather than create a new copy of the file for each version because many consumer files are large (e.g., multimedia files) and are only partially modified (e.g., by updating an ID3 header). Modifications that delete files such as `unlink` and `rename` cause the current version of the file to be saved as a log checkpoint.

4.4.4 Cache policies

Our final two policies control the caching of quFile data in the BlueFS on-disk cache. For a distributed file system, the decision of what files to cache locally significantly impacts user experience when disconnected.

quFiles may optionally specify two cache policies. A `cache insert` policy is called when a quFile is read and may specify which of its contents to cache on disk. Files specified by the cache insert policy are kept on a per-cache list by the BlueFS daemon and are fetched and stored when the daemon periodically prefetches data for the cache. For instance, when a quFile containing the recent episode of a favorite TV show is prefetched to a portable video player, its `cache insert` policy might specify that the video formatted for the video player, a representation that resides in that quFile, should also be prefetched. In contrast, when the same policy runs on a laptop, it would specify that the full-quality video should be fetched and cached instead. Thus, the policy ensures that only the data needed to play the video on each device is actually cached on the device's disk.

A `cache eviction` policy is called when the file system needs to reclaim disk space. The policy specifies whether or not cached contents should be evicted. Cache policies complement type-specific caching mechanisms in mobile storage systems [29, 34, 36] by adding the ability to make cache decisions based on dynamic context such as battery state or location.

4.5 Context library

Through the Sprocket interface, quFiles have read-only access to all information available to the BlueFS daemon. Thus, in principle, policies can extract arbitrary user-level context information in order to determine which representations to return. However, for conve-

Function	Returns
getUserName	char* username
getUserGroupId	uid_t uid, gid_t gid
getProcessName	char* procname
getHostname	char* hostname
getOSname	char* osname
getOSversion	char* release, char* version
getMachine	char* family
getCPUvendor	char* vendor, char* model
getCPUspeed	double cpuSpeed
getCPUutil	double utilization
getMemUtil	double utilization
getPowerState	enum{A/C, Battery}
getLocation	double latitude, double longitude
getServerBandwidth	double bandwidth
getServerLatency	double latency

Table 1. quFile context library

nience, we have implemented a library against which policies may link. This library contains the functions shown in Table 1 that query commonly-used context.

4.6 File system requirements for quFiles

Since our current implementation leverages BlueFS, it is useful to consider what features of BlueFS would need to be supported by a file system before we could port quFiles to that file system. First, quFiles require a method to notify applications when files are created or modified. While OS-specific notification mechanisms such as Linux's inotify [23] would suffice for a local file system, BlueFS persistent queries are useful in that they allow notifications to be delivered to any client of the distributed file system. Second, quFiles require a method to isolate the execution of extensions. This could be as simple as a user-level daemon process, or we could leverage existing extensibility research [3, 10, 39]. Finally, quFiles reuse existing file system directory support, as defined by POSIX.

5 Case Studies

The best way to evaluate the effectiveness and generality of a new abstraction is to implement several systems that use that abstraction to perform different tasks. Thus, in this section, we describe six case studies that use quFiles to extend the functionality of the file system. We have used these quFile case studies within our research group. The primary author of the paper has used quFiles for the last 12 months, while others have used quFiles for the past 6 months.

5.1 Resource management

One of the primary responsibilities of an operating system is to manage system resources such as CPU, memory, network, storage and power. While several research projects have shown that context can be used to craft more effective policies, almost every new proposed policy has resulted in a new system being built [1, 8, 27].

quFiles simplify resource management in two ways. First, they execute policies in the file system — thus, developers need not create new middleware or modify applications or the operating system. Second, developers only need to write resource management policies; quFiles take care of the mechanism.

Our case study allows a mobile computer to save battery energy by utilizing its spare storage capacity. Music playback is one of the most popular applications on mobile devices. Most mobile devices store music in a lossy, compressed format, such as the mp3 format, to conserve storage space and reduce network transfer times. However, decoding compressed music files requires significantly more computational power than playing uncompressed versions. For instance, the experimental results in Section 6.6 show a battery lifetime cost of 4–11% across several mobile devices. Further, we conducted a small survey to determine the amount of unused storage on cell phones and mp3 players. 13 of 45 mp3 players were over half empty, 18 were 50–90% full, and 14 were over 90% full. 15 of 29 cell phones were over half empty, 10 were 50–90% full and 4 were over 90% full.

Our quFile uses the spare storage on a mobile computer to store uncompressed versions of music files and then transparently provides those uncompressed version to music players to save energy. We built a quFile-aware transcoder that is notified when a new mp3 file is added to the distributed file system. The transcoder generates an uncompressed version of the music file with the same audio quality as the original, creates a quFile, links it to our policies, and moves both the compressed and uncompressed versions of the music file into the quFile using its raw view. Since persistent queries provide the ability to run the transcoder on any BlueFS client, we generate alternate transcodings on a wall-powered desktop computer. This shows one benefit of statically storing alternate representations in a quFile rather than generating them on-demand: we can avoid performing work on a resource-constrained device. In contrast, dynamically generating transcodings on a mobile device could substantially drain its battery.

The quFile `cache` policies ensure that only otherwise unused storage space is used to store uncompressed versions of music files. Using the normal BlueFS mechanisms, a music file is cached on a client either when it is first played or when it is prefetched by a user-specified policy (e.g., that all music files should be cached on a cell phone [29]). Since the music file is contained within a quFile, the file system's `lookup` function must always read the quFile before reading the music file. At this time, the quFile's `cache insert` policy is run. The policy queries the amount of storage space available on the device and adds the uncompressed representation to the prefetch list if space is available.

Later, when BlueFS does a regularly-scheduled prefetch of files for the mobile client, it retrieves files on

the prefetch list from the server if the mobile computer is plugged in, has spare storage available, and has network connectivity to the server. It adds these prefetched files to its on-disk cache. When BlueFS needs to evict files from the cache, it executes the quFile's `cache eviction` policy, which specifies that the uncompressed version is always evicted before any other data in the cache.

The `name` and `content` policies return the name and data for the uncompressed version of the music file if the mobile device is operating on battery power and the uncompressed version is cached on local storage, thereby improving battery lifetime. If the uncompressed version is not cached on the device, the original file is returned.

This case study demonstrates how quFiles achieve application and user transparency. All actions described above run automatically, without explicit user involvement and without application modification.

5.2 Versioning: a copy-on-write file system

Copy-on-write file systems such as Elephant [37] and ext3cow [31] create and retain previous versions of files when they are modified. Users can examine previous versions and revert the current version to a past one when desired. However, these systems are monolithic implementations, and the need to use new file systems has hindered their adoption. Thus, we were curious to see if quFiles could be used to add copy-on-write functionality to an existing file system.

We created a copy-on-write quFile that adds the ability to retain past versions of files. A user may choose to version any individual file, all files of a certain type, or all files in a particular subtree of the file system. For instance, a user might version all LaTeX source files. A quFile-aware utility uses BlueFS persistent queries to register for notifications when a file with the extension `.tex` is created. When it receives a notification, e.g., that `foo.tex` is being created, it creates a new quFile with the name `foo.tex.quFile`. It then uses the quFile's raw view to move the LaTeX file into the quFile and link the quFile to the copy-on-write policies.

In addition to the current version of the file, each copy-on-write quFile may contain possibly many older versions of the file. A past version may be represented as either a *checkpoint*, which is a complete past version of the file, or a *reverse delta*, which captures only the changes needed to reconstruct that version from the next most recent one. The reverse delta scheme is effectively an undo log that reduces the storage space needed to store past data; for instance, a change to the header of a 1 GB video file can be represented by a delta file only one block in size. While reverse deltas save storage, generating a complete copy of a past version incurs additional latency when one or more deltas are applied to a checkpoint or the current version.

The quFile's `name` and `content` policies simply return the current version of the file for the default view. The quFile's edit policy specifies that a new version should be created on any modification, i.e., whenever a file is closed, deleted, or renamed. Thus, when the user opens a file and issues one or more writes, the old data needed to undo his changes are saved to a new delta file within the quFile. The modifications are written to the current version of the file stored within the quFile. Because the default view exposes only the current version, these actions and the presence of past versions are completely transparent.

Versioning the data overwritten by file writes often consumes less storage and takes less time than creating a full checkpoint. To further reduce the cost of versioning, quFiles create new versions at the granularity of file `open` and `close` operations, rather than at each individual write. Unlike `write`, operations such as `rename` and `unlink` affect the entire file. For these operations, the current version is moved to a checkpoint within the quFile. Since there is no current version remaining, the quFile's `name` policy does not return a filename for the default view, giving the appearance that the file has been deleted. However, the old data can still be accessed via the raw view or a custom view.

When the user wishes to view prior versions, she uses the `versions` custom view (the `.quFile.versions` extension). This allows the use of COTS applications such as `ls` and graphical file system browsers to view versions. Whereas the default view only shows a single file, `foo.tex`, in a directory, the custom view may additionally show several past versions, e.g., `foo.tex`, `foo.tex.ckpt.monday`, `foo.tex.ckpt.last_week`, etc. When the `name` policy receives the `versions` keyword, it returns the names of any past versions found in the quFile's undo log. A user may use the `versions` keyword to specify all versions within a subtree; for example, `grep bar -Rn src.quFile.versions` searches for bar in all versions of all files in all subdirectories of `src`.

To conserve storage space, we dynamically generate checkpoints of past versions when they are viewed using the `versions` view. The quFile's `content` policy receives one of the names returned by the `name` policy. It dynamically creates a new checkpoint file within the quFile by applying the reverse deltas in succession to the next most recent checkpoint or the current version of the file. In addition to saving storage space, dynamic resolution also saves work in the common case where the user never inspects a past version. The performance hit of instantiating a previous checkpoint is taken only in the uncommon case when a user recovers a past version.

We have also implemented a quFile-aware garbage collection utility that runs as a cron job and removes older versions to save disk space. One sample policy maintains all prior versions less than one day old, one

version from the previous day, one from the prior two days, and one additional version from each exponentially increasing number of days.

5.3 Availability: resource-aware directories

Distributed file systems typically make no visible distinction between data cached locally and data that must be fetched from a remote server. Unfortunately, the absence of this distinction is often frustrating. For instance, a directory listing might reveal interesting multimedia content that the user tries to view. However, the user subsequently finds out that the content cannot be viewed satisfactorily because it is not cached locally and the network bandwidth to the server is insufficient to sustain the bit rate required to play the content.

To address this problem, we created a resource-aware directory listing policy that uses quFiles to tailor the contents of the directory to match the resources available to the computer. Our policy currently tailors directory listings to reflect cache state and network bandwidth. We can imagine similar policies that tailor listings to match the availability of CPU cycles or battery energy.

If a multimedia file is cached on a computer, the `name` policy's default view returns its name to the application. Otherwise, the policy returns the name of the multimedia file only if the network bandwidth to the server is greater than the bit rate needed to play the file.

The effect of the `name` policy is that a multimedia file is not displayed by directory listings or media players if there is insufficient network bandwidth to play it. Thus, a media player that is shuffling randomly among songs will not experience a glitch when it tries to play an unavailable song. A user will not have to experiment to find out which songs can be played and which cannot.

However, our experience using this policy revealed that sometimes we want to see files that are currently unavailable when we list a directory. For instance, a video player may support buffering, and we are willing to tolerate a delay before we watch a video. We therefore altered the `name` policy to support a custom view that simply changes the name of a file from `foo` to `foo_is_currently_unavailable` when the file is unplayable. The custom view is selected using the keyword `all`; e.g., `ls MyMusic.quFile.all` shows `foo_is_currently_unplayable`, while `ls MyMusic` does not show an entry for that file.

5.4 Security: context-aware data redaction

Mobile computers may be used at any location, including those that are insecure. For this reason, information scrubbing [19] has been proposed to protect, isolate and constrain private data on mobile devices. For instance, a user may not want to view her bank records or credit card information in a coffee shop or other public venue because others may observe personal or sensitive information by glancing at the screen. To help such users, we created a quFile that shows only redacted versions of files with sensitive data removed when data is viewed at insecure locations. The original data is displayed at secure locations.

This case study redacts only the presentation of data, not the bytes stored on disk. Thus, it guards against inadvertent display of data on a mobile computer, but not against the computer being lost or stolen.

We first created a quFile-aware utility that redacts XML files containing sensitive data. This utility is notified when files that may contain sensitive data are added to the file system. While our utility can redact any XML file using type-specific rules, we currently use it only for GnuCash, a personal finance program that stores data in a binary XML format. GnuCash [16] runs on Linux and is compatible with the Quicken Interchange Format.

Our utility parses each GnuCash file and generates a redacted version. The general-purpose redactor uses the Xerces [41] XML parser to apply type-specific transformation rules that obfuscate sensitive data. Our current rules obfuscate details such as account numbers, transaction details and dates, but leave the balances visible. Finally, the utility creates a quFile and moves both the original and redacted files into the quFile using its raw view. The redactor generates these two static representations each time the file is modified.

When an application reads this quFile, our context-aware declassification policy determines the location of the mobile computer using a modified version of Place Lab [25, 40]. If the computer is at a trusted location, as specified by a configuration file, the original version is returned. Otherwise the redacted version is displayed. Since the file type of the original and redacted versions are the same, the `name` policy returns the same name in all locations; however the data returned by the `content` policy may change as the user moves.

We did not need to modify GnuCash since it uses the transparent default view. GnuCash simply displays the original or redacted values in its GUI, depending on the location of the mobile computer. A quFile-aware user may override the `content` policy and view a different version using the quFile's raw view; e.g., by specifying `/bluefs/credit_card.quFile/credit_card.xml` instead of `/bluefs/credit_card.xml`.

5.5 Application-aware adaptation: Odyssey

Odyssey [27] introduced the notion of application-aware adaptation, in which the operating system monitors resource availability and notifies applications of any relevant changes. When notified by Odyssey of a resource level change, applications adjust the fidelity of the data they consume. A drawback of Odyssey is that both the operating system and applications must be modified. However, we observe that almost all application modifi-

cation is due to implementing the adaptation policy and mechanism inside the application. Thus, we decided to re-implement the functionality of Odyssey using quFiles. Unlike Odyssey, our quFile implementation requires no application modification. The adaptation policy can be removed from the application and cleanly specified using the quFile interface.

Our Odyssey implementation replicates Odyssey's Web (image viewing) application. A similar policy could be used for other Odyssey data types such as speech, maps [11], and 3-D graphics [24].

We created a utility that is notified when new JPEG images such as photos are added to the file system. The utility creates four additional lower-fidelity representations of the photo with varying JPEG quality levels. It creates a quFile, links in our Odyssey policies, and moves the lower-fidelity representations and the original image into the quFile using its raw view.

When a photo viewer lists a directory containing an image quFile, the Odyssey `name` policy returns the name of the original image file. However, when the content of the image is read, the quFile's `content` policy returns the best quality representation that can be displayed within one second.

The `content` policy uses the context library to determine the client's current bandwidth to the server. It reads the size of each representation in the quFile starting with the highest-fidelity, original representation and proceeding to the lowest. If a representation is cached locally or can be fetched from the server in less than a second, the `content` policy returns the inode for that representation. If no representation can meet the service time requirement, the lowest fidelity representation is returned.

The `edit` policy returns a context-specific value. It allows all modifications to the original image since the quFile-aware transcoder will be notified to regenerate alternate representations from the modified original. However, the policy disallows modifications to multimedia data in low-fidelity representations because it is unclear how such modifications can be reflected back to the original and other representations. This behavior is similar to the one users see in other arenas (e.g., when they try to save an Office document in a reduced-fidelity format such as ASCII text).

After experimenting with this policy, we made two further refinements. First, we realized that most edits to multimedia files change only the metadata header, which is identical across formats and quality levels. Thus, we modified our policy to allow editing of metadata for low-fidelity representations. The transcoder propagates metadata changes to other representations.

We also realized that some image editors rewrite the entire image instead of just modifying its metadata. We therefore modified our `edit` policy to allow writes outside the metadata region if the data written is identical to the data in the file. With these changes, all edits we attempted to make to low-fidelity versions succeeded. Of course, this is just one policy, and different applications may craft other policies such as allowing edits to low-fidelity data or creating multiple versions.

5.6 Platform-specific video display

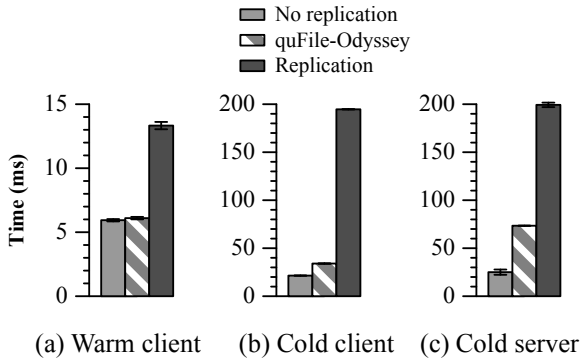
Section 4.1 gave a brief overview of our last case study, which transcodes videos to meet the resource constraints of file system clients. The authors currently use TiVo DVRs, N800 Internet tablets, and laptop computers to display videos. When a new `.TiVo` file is recorded and stored in BlueFS, a quFile-aware utility generates a full-resolution `.mp4` for the laptop and a lower-fidelity `.mp4` representation for the Nokia N800. Since the N800 has a lower screen resolution, we can save storage space on that device by producing a video formatted specifically for the N800's smaller display. The utility creates a quFile and populates it with the original and transcoded videos for each computer type described above. If we were to use additional types of clients, our transcoder could produce versions for those devices.

The `name` and `content` policies query the machine type on which they are running using the context library described in Section 4.5. The `name` policy returns a name ending with `.TiVo` when the video is read by the DVR, as determined by seeing that the name of the requesting application is a TiVo-specific utility. Otherwise, the `name` policy returns a name ending with `.mp4`. The `content` policy determines the type of client using the context library and returns the encoding appropriate for that type. The `cache_insert` policy ensures that each device only caches the video encoding it will display. We use BlueFS' type-specific affinity to prefetch such encodings to each device. quFiles hide this manipulation from video display applications, which therefore do not need to be modified. In practice, we found that this cached store of videos on the N800 made many a bus-ride more enjoyable! We also implemented a simple eviction policy: when the device is running out of storage space, all prefetched recordings are deleted before content the user has explicitly cached.

6 Evaluation

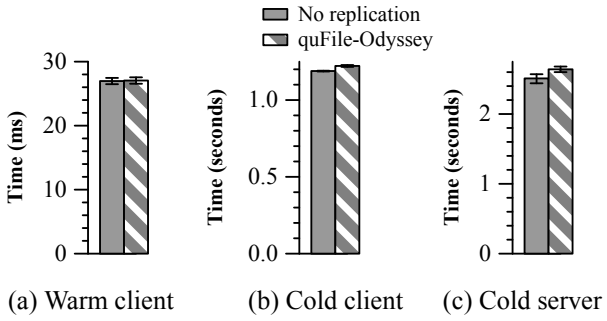
While the case studies in the previous section illustrate the generality of quFiles, we also verified that quFiles do not add too much overhead to file system operations and that the amount of code required to implement quFile policies is reasonable.

Unless otherwise stated, we evaluated quFiles on a Dell GX620 desktop with a 3.4 GHz Pentium 4 processor and 3 GB of DRAM. The desktop runs Ubuntu Linux 8.04 (Linux kernel 2.6.24). The desktop runs both the BlueFS server and client, and the BlueFS client does not use a local disk cache.



Each value is the mean of 10 trials; error bars are 90% confidence intervals. Note that the scales of the three graphs differ.

Figure 2. Time to list a directory with 100 images



Each value is the mean of 10 trials; error bars are 90% confidence intervals. Note that the scales of the three graphs differ.

Figure 3. Time to read 100 images

We executed each experiment in three scenarios. In the *warm client* scenario, the kernel's page cache contains all BlueFS data read during the experiment (the working sets of all experiments fit in memory). In the *cold client* scenario, no client data is in the kernel's page cache, but all server data is initially in the page cache. Thus, the first time an application reads a file page or attributes, an RPC is made to the server but no disk access is required. In the *cold server* scenario, no data is initially in any cache. On the first read, an RPC and a disk access are required to retrieve the data.

6.1 Directory listing

Our first experiment evaluates the performance overhead of quFiles for common file system operations by measuring the time to list the files in a directory and their attributes with the command `ls -al`. This is a worst-case scenario for using quFiles since the listing incurs the overhead of retrieving a quFile and executing both the `name` and `content` policies to determine which attributes to return for each file. Yet, there is minimal additional work to amortize this overhead because the directory listing requires that only the attributes of the file being listed be retrieved.

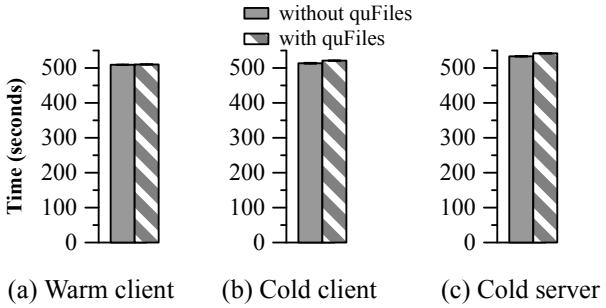
In our experiment, a directory contains 100 JPEG images. Each image is placed in a quFile that contains 4 additional low-fidelity representations and returns the appropriate one for the available server bandwidth using the Odyssey policy in Section 5.5.

The first bar for each scenario in Figure 2 shows a lower performance bound generated by assuming that Odyssey-like functionality is completely unsupported. Each value shows the time to list a directory without quFiles that contains only the original 100 JPEG images.

The second bar in each scenario shows the time to list the directory using quFiles. The Odyssey `name` and `content` policies return the name and content of the original image since server bandwidth is abundant. If the client cache is warm (which we expect to be the common case for most file system operations), quFiles add less than 3% overhead for this experiment (roughly 1.6 μs per file). If the client cache is cold, quFiles add 59% overhead. For each file, quFiles execute two policies. There is a measured overhead of 28 μs per policy, almost entirely due to user-level sandboxing. An additional 70 μs per file is required to fetch quFile attributes and contents from the server. If both the client and server caches are cold, the server performs two disk reads per file to read the quFile attributes and data. In this case, quFiles impose slightly less than a 3x overhead because disk reads are the dominant cost and three reads per file are performed with quFiles while only one read is performed without quFiles. However, it should be noted that even when both caches are cold, quFiles impose only 0.48 ms of overhead per file in this worst-case scenario. Note that the relative overhead of quFiles would decrease if file accesses were more random since, as directories, quFiles can be placed on disk near the files they contain (minimizing seeks).

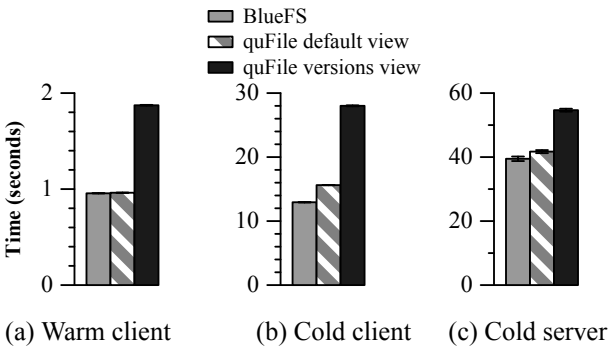
While the first bar in each scenario in the figure provides a lower bound on performance, a fairer comparison for Odyssey with quFiles is one in which all representations are stored together in the same directory. Odyssey uses this storage method for video, map, and speech data [27, 11]. Thus, there are 500 files in the directory. As the last bar in each scenario in Figure 2 shows, listing the directory takes over twice as long without quFiles in the warm client and cold server scenarios, and over 5 times as long in the cold client scenario. Because each quFile encapsulates many representations but returns only one, quFiles fetch less data than a regular file system when a naive storage layout policy is used.

Overall, we conclude that quFiles add minimal overhead to common file system operations, especially when the client cache is warm. Compared to naive file system layouts, quFiles can sometimes improve performance through their encapsulation properties.



Each value is the mean of 5 trials; error bars are 90% confidence intervals.

Figure 4. Time to make the Linux kernel



Each value is the mean of 5 trials; error bars are 90% confidence intervals. Note that the scales of the three graphs differ.

Figure 5. Time to search through the Linux kernel

6.2 Reading data

Often, users and applications will read file data, not just file attributes. We therefore ran a second microbenchmark that measures the time taken by the `cat` utility to read all images in our test directory and pipe the output to */dev/null*. As Figure 3 shows, quFile overhead is negligible in the warm client scenario, 3% in the cold client scenario, and 5% in the cold server scenario. Although the total overhead of quFile indirection remains the same as in the previous experiment, that overhead is now amortized across more file system activity. Thus, relative overhead decreases substantially.

6.3 Andrew-style make benchmark

We next turned our attention to application-level benchmarks. We started with a benchmark that measures quFile overhead during a complete `make` of the Linux 2.6.24-2 kernel. Such benchmarks, while perhaps not representative of modern workloads, have long been used to stress file system performance [18].

We compare the time to build the Linux kernel on BlueFS with and without quFiles. For the quFile test, we created a kernel source tree in which all source files (ending in .c, .h, or .S) are versioned using the copy-on-write quFile described in Section 5.2. The kernel source tree contains 23,062 files, of which 19,844 are versioned. Each quFile contains the original file and a checkpoint of approximately the same size as the original.

As Figure 4 shows, quFiles add negligible overhead in the warm client scenario and 1% overhead in the cold client and cold server scenarios. Even though kernel source files are quite small (averaging 11,663 bytes per file), many files such as headers are read multiple times, meaning that the extra overhead of fetching quFile data from the server can be amortized across multiple file reads. Further, computation is a significant portion of this benchmark, reducing the performance impact of I/O.

6.4 Kernel grep

We next ran a read-only benchmark that stresses file I/O performance. We used `grep` to search through the Linux source tree described in the previous section to find all 9 occurrences of "remove_wait_queue_locked".

The first bar in each scenario of Figure 5 shows the time to search through the Linux source without quFiles. The second bar in each scenario shows the time to search through the source with quFiles using the default view. In this case, each quFile returns only the current version of each source file. Thus, the results returned by the two `grep` commands are identical.

In the warm client scenario, the performance of `grep` with quFiles is within 1% of the performance without quFiles. As we would expect, the overhead is larger when there is no data in the client cache: 21% in the cold cache scenario and 6% in the cold server scenario.

quFiles, however, allow greater functionality than a regular file system. For instance, we can search through not only the current versions of source files but also all past versions by simply executing `grep -Rn linux.quFile.versions` where `linux` is the root of the kernel source tree. This command, which uses the `versions` view of the copy-on-write quFile, searches through twice as much data and returns 18 matches.

The last bar in each scenario shows the time to execute `grep` using the `versions` view. Since approximately twice as much data is read, the version-aware search takes approximately twice as long as a search using the default view in the warm client scenario. However, in the cold server scenario, the search takes only 31% longer since quFile representations are located close to each other on disk, reducing seek times.

This scenario shows that even when there is little data or computation across which to amortize overhead, performance is still reasonable, especially when data resides in the kernel's page cache. Further, quFiles enable functionality that is unavailable using regular file systems.

6.5 Code size

We measure the effort required to develop new policies by counting the lines of code for the quFiles used in

Component	Name	Content	Edit	Cache	Total
Resource mgmt.	32	18	8	36	94
Versioning	29	18	8	n/a	55
Security	20	33	8	n/a	61
Availability	64	26	8	n/a	98
Odyssey	23	27	32	n/a	82
Platform spec.	31	30	8	43	112

Table 2. Lines of code for quFile policies

each of our six case studies. As Table 2 shows, almost all policies required less than 100 lines of code. Compared to the code size of their monolithic ancestors, these numbers represent a dramatic reduction. For instance, the base Odyssey source is comprised of 32,329 lines of code while ext3cow requires a 18,494 line patch to the Linux-2.6.20.3 source tree. Our quFile implementation added 1,515 lines of code to BlueFS (BlueFS has 28,788 lines of code without quFiles). Further, all policies were implemented by a single graduate student. All policies took less than two weeks to implement. Later policies required only a few days as we gained experience.

6.6 Energy saving results

To evaluate the effectiveness of our case study in Section 5.1 that plays uncompressed music files to save energy, we measured the power used to play the uncompressed version of music files returned by quFiles and the power used to play the equivalent mp3 files. Table 3 shows results for three mobile devices: an HP4700 iPAQ handheld and Nokia N95-1 and N95-3 smart phones. The iPAQ runs Familiar v8.4, with OpiePlayer as its media player while the the N95-1 and N95-3 ran their factory-installed operating system and media players.

We directly measured the power consumed on the iPAQ by removing its battery and connecting its power supply cable through a digital multimeter. Unfortunately, the Nokia smart phones cannot operate with their battery unplugged, so we instead used the Nokia Energy Profiler [28] to measure playback power. Our tests show that quFiles can increase the battery lifetime of these devices by 4–11% when they are playing music. Given the importance of battery lifetime for these devices, this is a nice gain, especially considering that only spare resources are used to achieve it.

7 Conclusion

The quFile abstraction simplifies data management by providing a common mechanism for selecting one of several possible representations of the same logical data depending on the context in which it is accessed. A quFile also encapsulates the messy details of generating and storing multiple representations and the policies for selecting among them. We have shown the generality of quFiles by implementing six case studies that use them.

Device	Power to play mp3 files (mW)	Power with quFiles (mW)	Battery life extension
HP4700 iPAQ	1549	1401	11%
Nokia N95-1	962	914	5%
Nokia N95-3	454	437	4%

This table compares the power used to play mp3 files on 3 mobile devices with the power required to play the uncompressed versions returned by quFiles.

Table 3. Power savings enabled by quFiles

Acknowledgments

We thank Mona Attariyan, Dan Peek, Doug Terry, Benji Wester, our shepherd Karsten Schwan, and the anonymous reviewers for comments that improved this paper. We used David A. Wheeler's SLOCCount to estimate the lines of code for our implementation. Jason Flinn is supported by NSF CAREER award CNS-0346686. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, the University of Michigan, Microsoft, or the U.S. government.

References

[1] ANAND, M., NIGHTINGALE, E. B., AND FLINN, J. Self-tuning wireless network power management. In *Proceedings of the 9th Annual Conference on Mobile Computing and Networking* (San Diego, CA, September 2003), pp. 176–189.

[2] BELARAMANI, N., DAHLIN, M., GAO, L., NAYATE, A., VENKATARAMANI, A., YALAGANDULA, P., AND ZHENG, J. PRACTI Replication. In *Proceedings of the 3rd Symposium on Networked System Design and Implementation* (San Jose, CA, May 2006), pp. 59–72.

[3] BERSHAD, B., SAVAGE, S., PARDYAK, P., SIRER, E., FIUCZYNSKI, M., BECKER, D., CHAMBERS, C., AND EGGERS, S. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Copper Mountain, CO, Dec. 1995), pp. 267–284.

[4] BERSHAD, B. B., AND PINKERTON, C. B. Watchdogs - extending the UNIX file system. *Computer Systems 1*, 2 (Spring 1988).

[5] BILA, N., RONDA, T., MOHOMED, I., TRUONG, K. N., AND DE LARA, E. PageTailor: Reusable end-user customization for the mobile web. In *Proceedings of the 5th International Conference on Mobile Systems, Applications and Services* (San Juan, Puerto Rico, June 2007), pp. 16–29.

[6] Bundle programming guide. `http://developer.apple.com/documentation/CoreFoundation/Conceptual/CFBundles/CFBundles.html`.

[7] DE LARA, E., KUMAR, R., WALLACH, D. S., AND ZWAENEPOEL, W. Collaboration and multimedia authoring on mobile devices. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services* (San Francisco, CA, May 2003), pp. 287–301.

[8] DE LARA, E., WALLACH, D. S., AND ZWAENEPOEL, W. Puppeteer: Component-based adaptation for mobile computing. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems* (San Francisco, CA, March 2001), pp. 159–170.

[9] DOURISH, P., EDWARDS, W. K., LAMARCA, A., LAMPING, J., PETERSEN, K., SALISBURY, M., TERRY, D. B., AND THORNTON, J. Extending document management systems with user-specific active properties. *ACM Transactions on Information Systems 18*, 2 (2000), 140–170.

[10] ENGLER, D., KAASHOEK, M., AND J. O'TOOLE, J. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Copper Mountain, CO, December 1995), pp. 251–266.

[11] FLINN, J., AND SATYANARAYANAN, M. Energy-aware adaptation for mobile applications. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles* (Kiawah Island, SC, December 1999), pp. 48–63.

[12] FOX, A., GRIBBLE, S. D., BREWER, E. A., AND AMIR, E. Adapting to network and client variability via on-demand dynamic distillation. In *Proceedings of the 7th International ACM Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, October 1996), pp. 160–170.

[13] Filesystem in Userspace. `http://fuse.sourceforge.net/`.

[14] GEHANI, N. H., JAGADISH, H. V., AND ROOME, W. D. OdeFS: A file system interface to an object-oriented database. In *Proceedings of the 20th International Conference on Very Large Databases* (Santiago de Chile, Chile, September 1994), pp. 249–260.

[15] GIFFORD, D. K., JOUVELOT, P., SHELDON, M. A., AND O'TOOLE, J. W. Semantic file systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (Pacific Grove, CA, October 1991), pp. 16–25.

[16] GnuCash: Free Accounting Software. `http://www.gnucash.org`.

[17] GUPTA, A., AND MUMICK, I. S. Maintenance of materialized views: Problems, techniques and applications. *IEEE Quarterly Bulletin on Data Engineering; Special Issue on Materialized Views and Data Warehousing 18*, 2 (1995), 3–18.

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

[19] IOANNIDIS, S., SIDIROGLOU, S., AND KEROMYTIS, A. D. Privacy as an operating system service. In *Proceedings of the 1st conference on USENIX Workshop on Hot Topics in Security* (Vancouver, B.C., Canada, 2006), pp. 45–50.

[20] KISTLER, J. J., AND SATYANARAYANAN, M. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems 10*, 1 (February 1992).

[21] KJÆR, K. A survey of context-aware middleware. In *Proceedings of the IASTED International Conference on Software Engineering* (Innsbruck, Austria, February 2007), pp. 148–155.

[22] LOPRESTI, D. P., AND LAWRENCE, S. A. Information leakage through document redaction: attacks and countermeasures. In *Proceedings of Document Recognition and Retrieval XII - International Symposium on Electronic Imaging* (San Jose, CA, January 2005), pp. 183–190.

[23] LOVE, R. Kernel Korner: Intro to inotify. *Linux Journal*, 139 (2005), 8.

[24] NARAYANAN, D., FLINN, J., AND SATYANARAYANAN, M. Using history to improve mobile application adaptation. In *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications* (Monterey, CA, August 2000), pp. 30–41.

[25] NICHOLSON, A. J., AND NOBLE, B. D. BreadCrumbs: Forecasting mobile connectivity. In *Proceedings of the 14th International Conference on Mobile Computing and Networking* (San Francisco, CA, September 2008), pp. 46–57.

[26] NIGHTINGALE, E. B., AND FLINN, J. Energy-efficiency and storage flexibility in the Blue File System. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 363–378.

[27] NOBLE, B. D., SATYANARAYANAN, M., NARAYANAN, D., TILTON, J. E., FLINN, J., AND WALKER, K. R. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles* (Saint-Malo, France, October 1997), pp. 276–287.

[28] NOKIA. Nokia Energy Profiler. `http://www.forum.nokia.com/main/resources/development_process/power_management/nokia_energy_profiler/`.

[29] PEEK, D., AND FLINN, J. EnsemBlue: Integrating distributed storage and consumer electronics. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, WA, November 2006), pp. 219–232.

[30] PEEK, D., NIGHTINGALE, E. B., HIGGINS, B. D., KUMAR, P., AND FLINN, J. Sprockets: Safe extensions for distributed file systems. In *Proceedings of the USENIX Annual Technical Conference* (Santa Clara, CA, June 2007), pp. 115–128.

[31] PETERSON, Z. N. J., AND BURNS, R. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transacations on Storage 1*, 2 (2005), 190–212.

[32] PHAN, T., ZORPAS, G., AND BAGRODIA, R. Middleware support for reconciling client updates and data transcoding. In *Proceedings of the 2nd International Conference on Mobile Systems, Applications and Services* (Boston, MA, 2004), pp. 139–152.

[33] PILLAI, P., KE, Y., AND CAMPBELL, J. Multi-fidelity storage. In *Proceedings of the ACM 2nd International Workshop on Video Surveillance and Sensor Networks* (New York, NY, 2004), pp. 72–79.

[34] RAMASUBRAMANIAN, V., RODEHEFFER, T. L., TERRY, D. B., WALRAED-SULLIVAN, M., WOBBER, T., MARSHALL, C. C., AND VAHDAT, A. Cimbiosys: A platform for content-based partial replication. In *Proceedings of the 6th Symposium on Networked System Design and Implementation* (Boston, MA, April 2009), pp. 261–276.

[35] RUSSINOVICH, M. E., AND SOLOMON, D. A. Advanced features of NTFS. *Microsoft Windows Internals* (2005), 719–721.

[36] SALMON, B., SCHLOSSER, S. W., CRANOR, L. F., AND GANGER, G. R. Perspective: Semantic data management for the home. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies* (San Francisco, CA, February 2009), pp. 167–182.

[37] SANTRY, D. S., FEELEY, M. J., HUTCHINSON, N. C., VEITCH, A. C., CARTON, R. W., AND OFIR, J. Deciding when to forget in the Elephant file system. *SIGOPS Operating Systems Review 33*, 5 (1999), 110–123.

[38] SCHILIT, B., ADAMS, N., AND WANT, R. Context-aware computing applications. In *IEEE Workshop on Mobile Computing Systems and Applications* (Santa Cruz, CA, 1994), pp. 85–90.

[39] SELTZER, M. I., ENDO, Y., SMALL, C., AND SMITH, K. A. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation* (Seattle, Washington, October 1996), pp. 213–227.

[40] SOHN, T., GRISWOLD, W. G., SCOTT, J., LAMARCA, A., CHAWATHE, Y., SMITH, I., AND CHEN, M. Experiences with Place Lab: an open source toolkit for location-aware computing. In *Proceedings of the 28th International Conference on Software Engineering* (Shanghai, China, May 2006), pp. 462–471.

[41] Xerces-C++ XML Parser. `http://xerces.apache.org/xerces-c/`.

[42] YUMEREFENDI, A. R., MICKLE, B., AND COX, L. P. TightLip: Keeping applications from spilling the beans. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation* (Cambridge, MA, April 2007), pp. 159–172.

Tracking Back References in a Write-Anywhere File System

Peter Macko
Harvard University
pmacko@eecs.harvard.edu

Margo Seltzer
Harvard University
margo@eecs.harvard.edu

Keith A. Smith
NetApp, Inc.
keith.smith@netapp.com

Abstract

Many file systems reorganize data on disk, for example to defragment storage, shrink volumes, or migrate data between different classes of storage. Advanced file system features such as snapshots, writable clones, and deduplication make these tasks complicated, as moving a single block may require finding and updating dozens, or even hundreds, of pointers to it.

We present *Backlog*, an efficient implementation of explicit *back references*, to address this problem. Back references are file system meta-data that map physical block numbers to the data objects that use them. We show that by using LSM-Trees and exploiting the write-anywhere behavior of modern file systems such as NetApp® WAFL® or btrfs, we can maintain back reference meta-data with minimal overhead (one extra disk I/O per 102 block operations) and provide excellent query performance for the common case of queries covering ranges of physically adjacent blocks.

1 Introduction

Today's file systems such as WAFL [12], btrfs [5], and ZFS [23] have moved beyond merely providing reliable storage to providing useful services, such as snapshots and deduplication. In the presence of these services, any data block can be referenced by multiple snapshots, multiple files, or even multiple offsets within a file. This complicates any operation that must efficiently determine the set of objects referencing a given block, for example when updating the pointers to a block that has moved during defragmentation or volume resizing. In this paper we present new file system structures and algorithms to facilitate such dynamic reorganization of file system data in the presence of block sharing.

In many problem domains, a layer of indirection provides a simple way to relocate objects in memory or on storage without updating any pointers held by users of the objects. Such virtualization would help with some of the use cases of interest, but it is insufficient for one of the most important—defragmentation.

Defragmentation can be a particularly important issue for file systems that implement block sharing to support snapshots, deduplication, and other features. While block sharing offers great savings in space efficiency, sub-file sharing of blocks necessarily introduces on-disk fragmentation. If two files share a subset of their blocks, it is impossible for both files to have a perfectly sequential on-disk layout.

Block sharing also makes it harder to optimize on-disk layout. When two files share blocks, defragmenting one file may hurt the layout of the other file. A better approach is to make reallocation decisions that are aware of block sharing relationships between files and can make more intelligent optimization decisions, such as prioritizing which files get defragmented, selectively breaking block sharing, or co-locating related files on the disk.

These decisions require that when we defragment a file, we determine its new layout in the context of other files with which it shares blocks. In other words, given the blocks in one file, we need to determine the other files that share those blocks. This is the key obstacle to using virtualization to enable block reallocation, as it would hide this mapping from physical blocks to the files that reference them. Thus we have sought a technique that will allow us to track, rather than hide, this mapping, while imposing minimal performance impact on common file operations. Our solution is to introduce and maintain *back references* in the file system.

Back references are meta-data that map physical block numbers to their containing objects. Such back references are essentially inverted indexes on the traditional file system meta-data that maps file offsets to physical blocks. The challenge in using back references to simplify maintenance operations, such as defragmentation, is in maintaining them efficiently.

We have designed Log-Structured Back References,

or *Backlog* for short, a write-optimized back reference implementation with small, predictable overhead that remains stable over time. Our approach requires no disk reads to update the back reference database on block allocation, reallocation, or deallocation. We buffer updates in main memory and efficiently apply them *en masse* to the on-disk database during file system consistency points (checkpoints). Maintaining back references in the presence of snapshot creation, cloning or deletion incurs no additional I/O overhead. We use database compaction to reclaim space occupied by records referencing deleted snapshots. The only time that we read data from disk is during data compaction, which is an infrequent activity, and in response to queries for which the data is not currently in memory.

We present a brief overview of write-anywhere file systems in Section 2. Section 3 outlines the use cases that motivate our work and describes some of the challenges of handling them in a write-anywhere file system. We describe our design in Section 4 and our implementation in Section 5. We evaluate the maintenance overheads and query performance in Section 6. We present related work in Section 7, discuss future work in Section 8, and conclude in Section 9.

2 Background

Our work focuses specifically on tracking back references in *write-anywhere* (or *no-overwrite*) file systems, such as btrfs [5] or WAFL [12]. The terminology across such file systems has not yet been standardized; in this work we use WAFL terminology unless stated otherwise.

Write-anywhere file systems can be conceptually modeled as trees [18]. Figure 1 depicts a file system tree rooted at the *volume root* or a *superblock*. Inodes are the immediate children of the root, and they in turn are parents of indirect blocks and/or data blocks. Many modern file systems also represent inodes, free space bitmaps, and other meta-data as hidden files (not shown in the figure), so every allocated block with the exception of the root has a parent inode.

Write-anywhere file systems never update a block in place. When overwriting a file, they write the new file data to newly allocated disk blocks, recursively updating the appropriate pointers in the parent blocks. Figure 2 illustrates this process. This recursive chain of updates is expensive if it occurs at every write, so the file system accumulates updates in memory and applies them all at once during a *consistency point* (CP or *checkpoint*). The file system writes the root node last, ensuring that it represents a consistent set of data structures. In the case of failure, the operating system is guaranteed to find a consistent file system state with contents as of the last CP. File systems that support journaling to stable storage

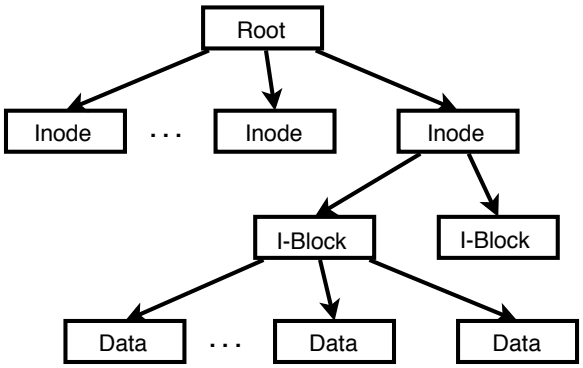


Figure 1: **File System as a Tree**. The conceptual view of a file system as a tree rooted at the *volume root* (superblock) [18], which is a parent of all inodes. An inode is a parent of data blocks and/or indirect blocks.

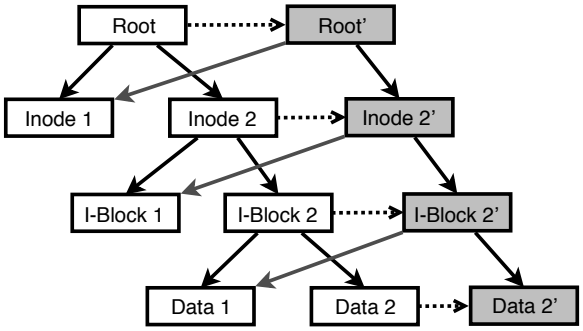


Figure 2: **Write-Anywhere file system maintenance**. In write-anywhere file systems, block updates generate new block copies. For example, upon updating the block "Data 2", the file system writes the new data to a new block and then recursively updates the blocks that point to it – all the way to the volume root.

(disk or NVRAM) can then recover data written since the last checkpoint by replaying the log.

Write-anywhere file systems can capture *snapshots*, point-in-time copies of previous file system states, by preserving the file system images from past consistency points. These snapshots are space efficient; the only differences between a snapshot and the live file system are the blocks that have changed since the snapshot copy was created. In essence, a write-anywhere allocation policy implements copy-on-write as a side effect of its normal operation.

Many systems preserve a limited number of the most recent consistency points, promoting some to hourly, daily, weekly, etc. *snapshots*. An asynchronous process typically reclaims space by deleting old CPs, reclaiming blocks whose only references were from deleted CPs. Several file systems, such as WAFL and ZFS, can create writable *clones* of snapshots, which are useful especially in development (such as creation of a writable

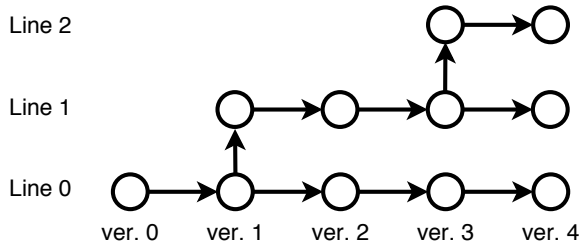


Figure 3: **Snapshot Lines**. The tuple (line, version), where *version* is a global CP number, uniquely identifies a snapshot or consistency point. Taking a consistency point creates a new version of the latest snapshot within each line, while creating a writable clone of an existing snapshot starts a new line.

duplicate for testing of a production database) and virtualization [9].

It is helpful to conceptualize a set of snapshots and consistency points in terms of *lines* as illustrated in Figure 3. A time-ordered set of snapshots of a file system forms a single *line*, while creation of a writable clone starts a new line. In this model, a (line ID, version) pair uniquely identifies a snapshot or a consistency point. In the rest of the paper, we use the global consistency point number during which a snapshot or consistency point was created as its version number.

The use of copy-on-write to implement snapshots and clones means that a single physical block may belong to multiple file system trees and have many meta-data blocks pointing to it. In Figure 2, for example, two different indirect blocks, *I-Block 2* and *I-Block 2'*, reference the block *Data 1*. Block-level deduplication [7, 17] can further increase the number of pointers to a block by allowing files containing identical data blocks to share a single on-disk copy of the block. This block sharing presents a challenge for file system management operations, such as defragmentation or data migration, that reorganize blocks on disk. If the file system moves a block, it will need to find and update all of the pointers to that block.

3 Use Cases

The goal of Backlog is to maintain meta-data that facilitates the dynamic movement and reorganization of data in write-anywhere file systems. We envision two major cases for internal data reorganization in a file system. The first is support for bulk data migration. This is useful when we need to move all of the data off of a device (or a portion of a device), such as when shrinking a volume or replacing hardware. The challenge here for traditional file system designs is translating from the physical block addresses we are moving to the files referencing those blocks so we can update their block pointers. Ext3, for example, can do this only by traversing the entire file system tree searching for block pointers that fall in the target range [2]. In a large file system, the I/O required for this brute-force approach is prohibitive.

Our second use case is the dynamic reorganization of on-disk data. This is traditionally thought of as defragmentation—reallocating files on-disk to achieve contiguous layout. We consider this use case more broadly to include tasks such as free space coalescing (to create contiguous expanses of free blocks for the efficient layout of new files) and the migration of individual files between different classes of storage in a file system.

To support these data movement functions in write-anywhere file systems, we must take into account the block sharing that emerges from features such as snapshots and clones, as well as from the deduplication of identical data blocks [7, 17]. This block sharing makes defragmentation both more important and more challenging than in traditional file system designs. Fragmentation is a natural consequence of block sharing; two files that share a subset of their blocks cannot both have an ideal sequential layout. And when we move a shared block during defragmentation, we face the challenge of finding and updating pointers in multiple files.

Consider a basic defragmentation scenario where we are trying to reallocate the blocks of a single file. This is simple to handle. We find the file's blocks by reading the indirect block tree for the file. Then we move the blocks to a new, contiguous, on-disk location, updating the pointer to each block as we move it.

But things are more complicated if we need to defragment two files that share one or more blocks, a case that might arise when multiple virtual machine images are cloned from a single master image. If we defragment the files one at a time, as described above, the shared blocks will ping-pong back and forth between the files as we defragment one and then the other. A better approach is to make reallocation decisions that are aware of the sharing relationship. There are multiple ways we might do this. We could select the most important file, and only optimize its layout. Or we could decide that performance is more important than space savings and make duplicate copies of the shared blocks to allow sequential layout for all of the files that use them. Or we might apply multi-dimensional layout techniques [20] to achieve near-optimal layouts for both files while still preserving block sharing.

The common theme in all of these approaches to layout optimization is that when we defragment a file, we must determine its new layout in the context of the other files with which it shares blocks. Thus we have sought a technique that will allow us to easily map physical blocks to the files that use them, while imposing minimal performance impact on common file system operations.

Our solution is to introduce and maintain *back reference* meta-data to explicitly track all of the logical owners of each physical data block.

4 Log-Structured Back References

Back references are updated significantly more frequently than they are queried; they must be updated on every block allocation, deallocation, or reallocation. It is crucial that they impose only a small performance overhead that does not increase with the age of the file system. Fortunately, it is not a requirement that the meta-data be space efficient, since disk is relatively inexpensive.

In this section, we present Log-Structured Back References (Backlog). We present our design in two parts. First, we present the conceptual design, which provides a simple model of back references and their use in querying. We then present a design that achieves the capabilities of the conceptual design efficiently.

4.1 Conceptual Design

A naïve approach to maintaining back references requires that we write a back reference record for every block at every consistency point. Such an approach would be prohibitively expensive both in terms of disk usage and performance overhead. Using the observation that a given block and its back references may remain unchanged for many consistency points, we improve upon this naïve representation by maintaining back references over ranges of CPs. We represent every such back reference as a record with the following fields:

- `block`: The physical block number
- `inode`: The inode number that references the block
- `offset`: The offset within the inode
- `line`: The line of snapshots that contains the inode
- `from`: The global CP number (time epoch) from which this record is valid (i.e., when the reference was allocated to the inode)
- `to`: The global CP number until which the record is valid (exclusive) or ∞ if the record is still alive

For example, the following table describes two blocks owned by inode 2, created at time 4 and truncated to one block at time 7:

block	inode	offset	line	from	to
100	2	0	0	4	∞
101	2	1	0	4	7

Although we present this representation as operating at the level of blocks, it can be extended to include a `length` field to operate on extents.

Let us now consider how a table of these records, indexed by physical block number, lets us answer the sort of query we encounter in file system maintenance. Imagine that we have previously run a deduplication process and found that many files contain a block of all 0's. We stored one copy of that block on disk and now have multiple inodes referencing that block. Now, let's assume that we wish to move the physical location of that block of 0's in order to shrink the size of the volume on which it lives. First we need to identify all the files that reference this block, so that when we relocate the block, we can update their meta-data to reference the new location. Thus, we wish to query the back references to answer the question, "Tell me all the objects containing this block." More generally, we may want to ask this query for a range of physical blocks. Such queries translate easily into indexed lookups on the structure described above. We use the physical block number as an index to locate all the records for the given physical block number. Those records identify all the objects that reference the block and all versions in which those blocks are valid.

Unfortunately, this representation, while elegantly simple, would perform abysmally. Consider what is required for common operations. Every block deallocation requires replacing the ∞ in the `to` field with the current CP number, translating into a read-modify-write on this table. Block allocation requires creating a new record, translating into an insert into the table. Block reallocation requires both a deallocation and an allocation, and thus a read-modify-write and an insert. We ran experiments with this approach and found that the file system slowed down to a crawl after only a few hundred consistency points. Providing back references with acceptable overhead during normal operation requires a feasible design that efficiently realizes the conceptual model described in this section.

4.2 Feasible Design

Observe that records in the conceptual table described in Section 4.1 are of two types. *Complete records* refer to blocks that are no longer part of the live file system; they exist only in snapshots. Such blocks are identified by having `to` $< \infty$. *Incomplete records* are part of the live file system and always have `to` $= \infty$. Our actual design maintains two separate tables, `From` and `To`. Both tables contain the first four columns of the conceptual table (`block`, `inode`, `offset`, and `line`). The `From` table also contains the `from` column, and the `To` table contains the `to` column. Incomplete records exist only in the `From` table, while complete records appear in both tables.

On a block allocation, regardless of whether the block is newly allocated or reallocated, we insert the corre-

sponding entry into the From table with the from field set to the current global CP number, creating an incomplete record. When a reference is removed, we insert the appropriate entry into the To table, completing the record. We buffer new records in memory, committing them to disk at the end of the current CP, which guarantees that all entries with the current global CP number are present in memory. This facilitates pruning records where from = to, which refer to block references that were added and removed within the same CP.

For example, the Conceptual table from the previous subsection (describing the two blocks of inode 2) is broken down as follows:

From:

block	inode	offset	line	from
100	2	0	0	4
101	2	1	0	4

To:

block	inode	offset	line	to
101	2	1	0	7

The record for block 101 is complete (has both From and To entries), while the record for 100 is incomplete (the block is currently allocated).

This design naturally handles block sharing arising from deduplication. When the file system detects that a newly written block is a duplicate of an existing on-disk block, it adds a pointer to that block and creates an entry in the From table corresponding to the new reference.

4.2.1 Joining the Tables

The conceptual table on which we want to query is the outer join of the From and To tables. A tuple F ∈ From joins with a tuple T ∈ To that has the same first four fields and that has the smallest value of T.to such that F.from < T.to. If there is a From entry without a matching To entry (i.e., a live, incomplete record), we outer-join it with an implicitly-present tuple T′ ∈ To with T′.to = ∞.

For example, assume that a file with inode 4 was created at time 10 with one block and then truncated at time 12. Then, the same block was assigned to the file at time 16, and the file was removed at time 20. Later on, the same block was allocated to a different file at time 30. These operations produce the following records:

From:

block	inode	offset	line	from
103	4	0	0	10
103	4	0	0	16
103	5	2	0	30

To:

block	inode	offset	line	to
103	4	0	0	12
103	4	0	0	20

Observe that the first From and the first To record form a logical pair describing a single interval during which the block was allocated to inode 4. To reconstruct the history of this block allocation, a record from = 10 has to join with to = 12. Similarly, the second From record should join with the second To record. The third From entry does not have a corresponding To entry, so it joins with an implicit entry with to = ∞.

The result of this outer join is the Conceptual view. Every tuple C ∈ Conceptual has both from and to fields, which together represent a range of global CP numbers within the given snapshot line, during which the specified block is referenced by the given inode from the given file offset. The range might include deleted consistency points or snapshots, so we must apply a mask of the set of valid versions before returning query results.

Coming back to our previous example, performing an outer join on these tables produces:

block	inode	offset	line	from	to
103	4	0	0	10	12
103	4	0	0	16	20
103	5	2	0	30	∞

This design is feasible until we introduce writable clones. In the rest of this section, we explain how we have to modify the conceptual view to address them. Then, in Section 5, we discuss how we realize this design efficiently.

4.2.2 Representing Writable Clones

Writable clones pose a challenge in realizing the conceptual design. Consider a snapshot (l, v), where l is the line and v is the version or CP. Naïvely creating a writable clone (l', v') requires that we duplicate all back references that include (l, v) (that is, C.line $= l \wedge$ C.from $\leq v <$ C.to, where C ∈ Conceptual), updating the line field to l' and the from and to fields to represent all versions (range $0 - \infty$). Using this technique, the conceptual table would continue to be the result of the outerjoin of the From and To tables, and we could express queries directly on the conceptual table. Unfortunately, this mass duplication is prohibitively expensive. Thus, our actual design cannot simply rely on the conceptual table. Instead we implicitly represent writable clones in the database using structural inheritance [6], a technique akin to copy-on-write. This avoids the massive duplication in the naïve approach.

The implicit representation assumes that every block of (l, v) is present in all subsequent versions of l', unless explicitly overridden. When we modify a block, b, in a new writable clone, we do two things: First, we declare the end of b's lifetime by writing an entry in the To table recording the current CP. Second, we record the alloca-

tion of the new block b' (a copy-on-write of b) by adding an entry into the From table.

For example, if the old block $b = 103$ was originally allocated at time 30 in line $l = 0$ and was replaced by a new block $b' = 107$ at time 43 in line $l' = 1$, the system produces the following records:

From:

block	inode	offset	line	from
103	5	2	0	30
107	5	2	1	43

To:

block	inode	offset	line	to
103	5	2	1	43

The entry in the To table *overrides* the inheritance from the previous snapshot; however, notice that this new To entry now has no element in the From table with which to join, since no entry in the From table exists with the line $l' = 1$. We join such entries with an implicit entry in the From table with from $= 0$. With the introduction of structural inheritance and implicit records in the From table, our joined table no longer matches our conceptual table. To distinguish the conceptual table from the actual result of the join, we call the join result the *Combined* table.

Summarizing, a back reference record C $\in$ Combined of (l, v) is implicitly present in all versions of l', unless there is an overriding record C′ $\in$ Combined with C.block $=$ C′.block $\wedge$ C.inode $=$ C′.inode $\wedge$ C.offset $=$ C′.offset $\wedge$ C′.line $=$ l′ $\wedge$ C′.from $= 0$. If such a C′ record exists, then it defines the versions of l' for which the back reference is valid (i.e., from C′.from to C′.to). The file system continues to maintain back references as usual by inserting the appropriate From and To records in response to allocation, deallocation and reallocation operations.

While the Combined table avoids the massive copy when creating writable clones, query execution becomes a bit more complicated. After extracting initial result from the Combined table, we must iteratively expand those results as follows. Let Initial be the initial result extracted from Combined containing all records that correspond to blocks $b_0, \ldots, b_n$. If any of the blocks b_i has one or more override records, they are all guaranteed to be in this initial result. We then initialize the query Result to contain all records in Initial and proceed as follows. For every record R $\in$ Result that references a snapshot (l, v) that was cloned to produce (l', v'), we check for the existence of a corresponding override record C′ $\in$ Initial with C′.line $= l'$. If no such record exists, we explicitly add records C′.line $\leftarrow l'$, C′.from $\leftarrow 0$ and C′.to $\leftarrow \infty$ to Result. This process repeats recursively until it fails to insert additional records. Finally, when the result is fully expanded we mask the ranges to remove references to deleted snapshots as described in Section 4.2.1.

This approach requires that we never delete the back references for a cloned snapshot. Consequently, snapshot deletion checks whether the snapshot has been cloned, and if it has, it adds the snapshot ID to the list of *zombies*, ensuring that its back references are not purged during maintenance. The file system is then free to proceed with snapshot deletion. Periodically we examine the list of zombies and drop snapshot IDs that have no remaining descendants (clones).

5 Implementation

With the feasible design in hand, we now turn towards the problem of efficiently realizing the design. First we discuss our implementation strategy and then discuss our on-disk data storage (section 5.1). We then proceed to discuss database compaction and maintenance (section 5.2), partitioning the tables (section 5.3), and recovering the tables after system failure (section 5.4). We implemented and evaluated the system in fsim, our custom file system simulator, and then replaced the native back reference support in btrfs with Backlog.

The implementation in fsim allows us to study the new feature in isolation from the rest of the file system. Thus, we fully realize the implementation of the back reference system, but embed it in a simulated file system rather than a real file system, allowing us to consider a broad range of file systems rather than a single specific implementation. Fsim simulates a write-anywhere file system with writable snapshots and deduplication. It exports an interface for creating, deleting, and writing to files, and an interface for managing snapshots, which are controlled either by a stochastic workload generator or an NFS trace player. It stores all file system meta-data in main memory, but it does not explicitly store any data blocks. It stores only the back reference meta-data on disk. Fsim also provides two parameters to configure deduplication emulation. The first specifies the percentage of newly created blocks that duplicate existing blocks. The second specifies the distribution of how those duplicate blocks are shared.

We implement back references as a set of callback functions on the following events: adding a block reference, removing a block reference, and taking a consistency point. The first two callbacks accumulate updates in main memory, while the consistency point callback writes the updates to stable storage, as described in the next section. We implement the equivalent of a user-level process to support database maintenance and query. We verify the correctness of our implementation by a utility program that walks the entire file system tree, reconstructs the back references, and then compares them with the database produced by our algorithm.

5.1 Data Storage and Maintenance

We store the `From` and `To` tables as well as the precomputed `Combined` table (if available) in a custom row-oriented database optimized for efficient insert and query. We use a variant of LSM-Trees [16] to hold the tables. The fundamental property of this structure is that it separates an in-memory *write store* (WS or C_0 in the LSM-Tree terminology) and an on-disk *read store* (RS or C_1).

We accumulate updates to each table in its respective WS, an in-memory balanced tree. Our `fsim` implementation uses a Berkeley DB 4.7.25 in-memory B-tree database [15], while our btrfs implementation uses Linux red/black trees, but any efficient indexing structure would work. During consistency point creation, we write the contents of the WS into the RS, an on-disk, densely packed B-tree, which uses our own LSM-Tree/Stepped-Merge implementation, described in the next section.

In the original LSM-Tree design, the system selects parts of the WS to write to disk and merges them with the corresponding parts of the RS (indiscriminately merging all nodes of the WS is too inefficient). We cannot use this approach, because we require that a consistency point has all accumulated updates persistent on disk. Our approach is thus more like the Stepped-Merge variant [13], in which the entire WS is written to a new RS run file, resulting in one RS file per consistency point. These RS files are called the Level 0 runs, which are periodically merged into Level 1 runs, and multiple Level 1 runs are merged to produce Level 2 runs, etc., until we get to a large Level N file, where N is fixed. The Stepped-Merge Method uses these intermediate levels to ensure that the sizes of the RS files are manageable. For the back references use case, we found it more practical to retain the Level 0 runs until we run data compaction (described in Section 5.2), at which point, we merge all existing Level 0 runs into a single RS (analogous to the Stepped-Merge Level N) and then begin accumulating new Level 0 files at subsequent CPs. We ensure that the individual files are of a manageable size using horizontal partitioning as described in Section 5.3.

Writing Level 0 RS files is efficient, since the records are already sorted in memory, which allows us to construct the compact B-tree bottom-up: The data records are packed densely into pages in the order they appear in the WS, creating a Leaf file. We then create an Internal 1 (I1) file, containing densely packed internal nodes containing references to each block in the Leaf file. We continue building I files until we have an I file with only a single block (the root of the B-tree). As we write the Leaf file, we incrementally build the I1 file and iteratively, as we write I file, In, to disk, we incrementally build the I$(n+1)$ file in memory, so that writing the I files requires no disk reads.

Queries specify a block or a range of blocks, and those blocks may be present in only some of the Level 0 RS files that accumulate between data compaction runs. To avoid many unnecessary accesses, the query system maintains a Bloom filter [3] on the RS files that is used to determine which, if any, RS files must be accessed. If the blocks are in the RS, then we position an iterator in the Leaf file on the first block in the query result and retrieve successive records until we have retrieved all the blocks necessary to satisfy the query.

The Bloom filter uses four hash functions, and its default size for `From` and `To` RS files depends on the maximum number of operations in a CP. We use 32 KB for 32,000 operations (a typical setting for WAFL), which results in an expected false positive rate of up to 2.4%. If an RS contains a smaller number of records, we appropriately shrink its Bloom filter to save memory. This operation is efficient, since a Bloom filter can be halved in size in linear time [4]. The default filter size is expandable up to 1 MB for a `Combined` read store. False positives for the latter filter grow with the size of the file system, but this is not a problem, because the `Combined` RS is involved in almost all queries anyway.

Each time that we remove a block reference, we prune in real time by checking whether the reference was both created and removed during the same interval between two consistency points. If it was, we avoiding creating records in the Combined table where `from` = `to`. If such a record exists in `From`, our buffering approach guarantees that the record resides in the in-memory WS from which it can be easily removed. Conversely, upon block reference addition, we check the in-memory WS for the existence of a corresponding `To` entry with the same CP number and proactively prune those if they exist (thus a reference that exists between CPs 3 and 4 and is then reallocated in CP 4 will be represented with a single entry in Combined with a lifespan beginning at 3 and continuing to the present). We implement the WS for all the tables as balanced trees sorted first by `block`, `inode`, `offset`, and `line`, and then by the `from` and/or `to` fields, so that it is efficient to perform this proactive pruning.

During normal operation, there is no need to delete tuples from the RS. The masking procedure described in Section 4.2.1 addresses blocks deleted due to snapshot removal.

During maintenance operations that relocate blocks, e.g., defragmentation or volume shrinking, it becomes necessary to remove blocks from the RS. Rather than modifying the RS directly, we borrow an idea from the C-store, column-oriented data manager [22] and retain a *deletion vector*, containing the set of entries that should not appear in the RS. We store this vector as a B-tree in-

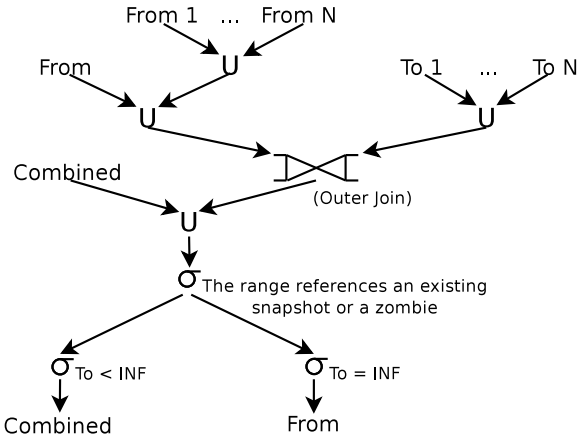


Figure 4: **Database Maintenance**. This query plan merges all on-disk RS's, represented by the "From N", precomputes the `Combined` table, which is the join of the `From` and `To` tables, and purges old records. Incomplete records reside in the on-disk `From` table.

dex, which is usually small enough to be entirely cached in memory. The query engine then filters records read from the RS according to the deletion vector in a manner that is completely opaque to query processing logic. If the deletion vector becomes sufficiently large, the system can optionally write a new copy of the RS with the deleted tuples removed.

5.2 Database Maintenance

The system periodically compacts the back reference indexes. This compaction merges the existing Level 0 RS's, precomputes the `Combined` table by joining the `From` and `To` tables, and purges records that refer to deleted checkpoints. Merging RS files is efficient, because all the tuples are sorted identically.

After compaction, we are left with one RS containing the complete records in the `Combined` table and one RS containing the incomplete records in the `From` table. Figure 4 depicts this compaction process.

5.3 Horizontal Partitioning

We partition the RS files by block number to ensure that each of the files is of a manageable size. We maintain a single WS per table, but then during a checkpoint, we write the contents of the WS to separate partitions, and compaction processes each partition separately. Note that this arrangement provides the compaction process the option of selectively compacting different partitions. In our current implementation, each partition corresponds to a fixed sequential range of block numbers.

There are several interesting alternatives for partitioning that we plan to explore in future work. We could start with a single partition and then use a threshold-based scheme, creating a new partition when an existing partition exceeds the threshold. A different approach that might better exploit parallelism would be to use hashed partitioning.

Partitioning can also allow us to exploit the parallelism found in today's storage servers: different partitions could reside on different disks or RAID groups and/or could be processed by different CPU cores in parallel.

5.4 Recovery

This back reference design depends on the write-anywhere nature of the file system for its consistency. At each consistency point, we write the WS's to disk and do not consider the CP complete until all the resulting RS's are safely on disk. When the system restarts after a failure, it is thus guaranteed that it finds a consistent file system with consistent back references at a state as of the last complete CP. If the file system has a journal, it can rebuild the WS's together with the other parts of the file system state as the system replays the journal.

6 Evaluation

Our goal is that back reference maintenance not interfere with normal file-system processing. Thus, maintaining the back reference database should have minimal overhead that remains stable over time. In addition, we want to confirm that query time is sufficiently low so that utilities such as volume shrinking can use them freely. Finally, although space overhead is not of primary concern, we want to ensure that we do not consume excessive disk space.

We evaluated our algorithm first on a synthetically generated workload that submits write requests as rapidly as possible. We then proceeded to evaluate our system using NFS traces; we present results using part of the EECS03 data set [10]. Next, we report performance for an implementation of Backlog ported into btrfs. Finally, we present query performance results.

6.1 Experimental Setup

We ran the first part of our evaluation in `fsim`. We configured the system to be representative of a common write-anywhere file system, WAFL [12]. Our simulation used 4 KB blocks and took a consistency point after every 32,000 block writes or 10 seconds, whichever came first (a common configuration of WAFL). We configured the deduplication parameters based on measure-

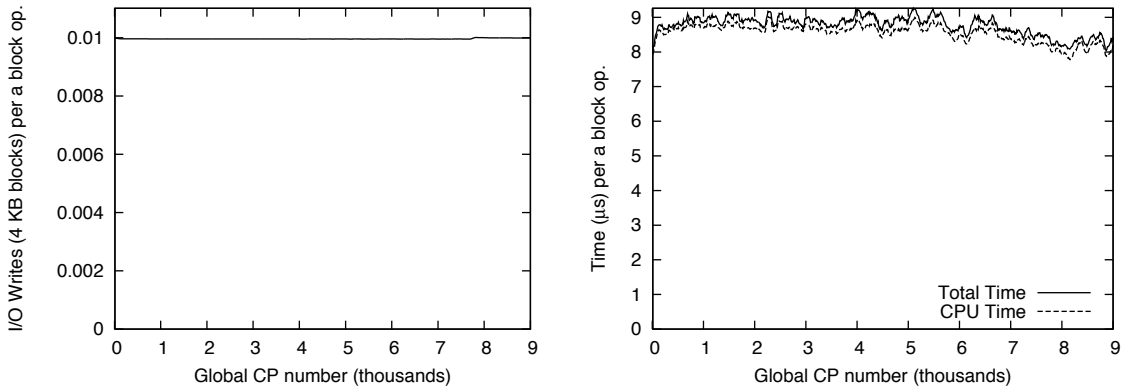


Figure 5: **Fsim Synthetic Workload Overhead during Normal Operation**. I/O overhead due to maintaining back references normalized per persistent block operations (adding or removing a reference with effects that survive at least one CP) and the time overhead normalized per block operation.

ments from a few file servers at NetApp. We treat 10% of incoming blocks as duplicates, resulting in a file system where approximately 75 – 78% of the blocks have reference counts of 1, 18% have reference counts of 2, 5% have reference counts of 3, etc. Our file system kept four hourly and four nightly snapshots.

We ran our simulations on a server with two dual-core Intel Xeon 3.0 GHz CPUs, 10 GB of RAM, running Linux 2.6.28. We stored the back reference meta-data from `fsim` on a 15K RPM Fujitsu MAX3073RC SAS drive that provides 60 MB/s of write throughput. For the micro-benchmarks, we used a 32 MB cache in addition to the memory consumed by the write stores and the Bloom filters.

We carried out the second part of our evaluation in a modified version of btrfs, in which we replaced the original implementation of back references by Backlog. As btrfs uses extent-based allocation, we added a `length` field to both the `From` and `To` described in Section 4.1. All fields in back reference records are 64-bit. The resulting `From` and `To` tuples are 40 bytes each, and a `Combined` tuple is 48 bytes long. All btrfs workloads were executed on an Intel Pentium 4 3.0 GHz, 512 MB RAM, running Linux 2.6.31.

6.2 Overhead

We evaluated the overhead of our algorithm in `fsim` using both synthetically generated workloads and NFS traces. We used the former to understand how our algorithm behaves under high system load and the latter to study lower, more realistic loads.

6.2.1 Synthetic Workload

We experimented with a number of different configurations and found that all of them produced similar results, so we selected one representative workload and used that throughout the rest of this section. We configured our workload generator to perform at least 32,000 block writes between two consistency points, which corresponds to the periods of high load on real systems. We set the rates of file create, delete, and update operations to mirror the rates observed in the EECS03 trace [10]. 90% of our files are small, reflecting what we observe on file systems containing mostly home directories of developers – which is similar to the file system from which the EECS03 trace was gathered. We also introduced creation and deletion of writable clones at a rate of approximately 7 clones per 100 CP's, although the original NFS trace did not have any analogous behavior. This is substantially more clone activity than we would expect in a home-directory workload such as EECS03, so it gives us a pessimal view of the overhead clones impose.

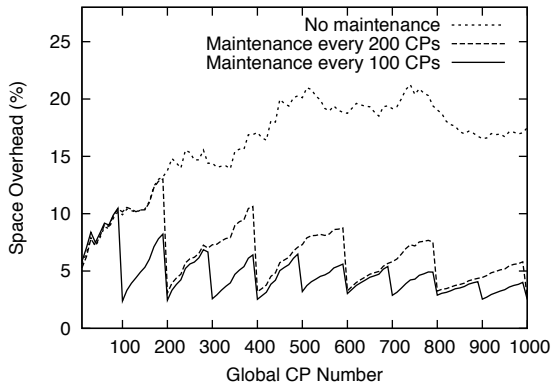


Figure 6: **Fsim Synthetic Workload Database Size**. The size of the back reference meta-data as a percentage of the total physical data size as it evolves over time. The disk usage at the end of the workload is 14.2 GB after deduplication.

Figure 5 shows how the overhead of maintaining back

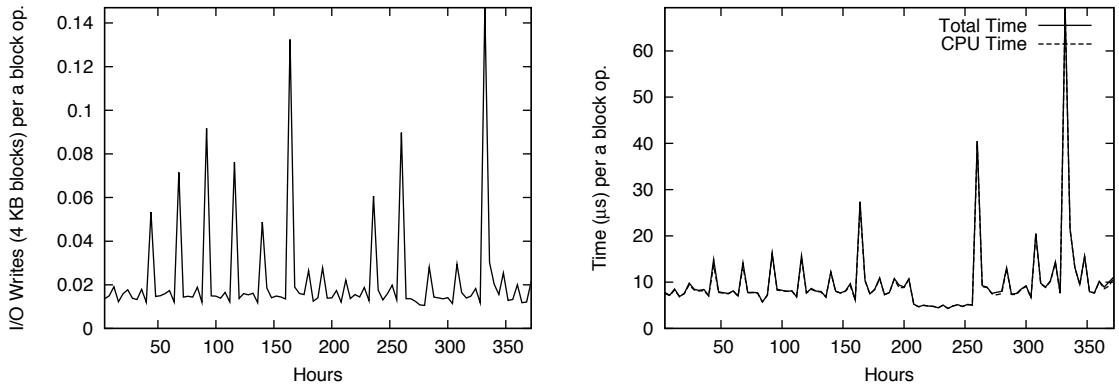


Figure 7: **Fsim NFS Trace Overhead during Normal Operation**. The I/O and time overheads for maintaining back references normalized per a block operation (adding or removing a reference).

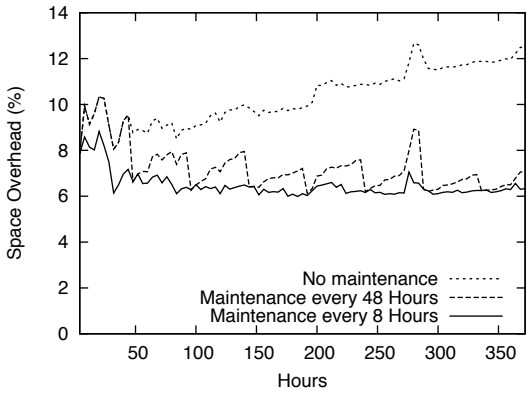


Figure 8: **Fsim NFS Traces: Space Overhead**. The size of the back reference meta-data as a percentage of the total physical data size as it evolves over time. The disk usage at the end of the workload is 11.0 GB after deduplication.

references changes over time, ignoring the cost of periodic database maintenance. The average cost of a block operation is 0.010 block writes or 8-9 μs per block operation, regardless of whether the operation is adding or removing a reference. A single copy-on-write operation (involving both adding and removing a block from an inode) adds on average 0.020 disk writes and at most 18 μs. This amounts to at most 628 additional writes and 0.5–0.6 seconds per CP. More than 95% of this overhead is CPU time, most of which is spent updating the write store. Most importantly, the overhead is stable over time, and the I/O cost is constant even as the total data on the file system increases.

Figure 6 illustrates meta-data size evolution as a percentage of the total physical data size for two frequencies of maintenance (every 100 or 200 CPs) and for no maintenance at all. The space overhead after maintenance drops consistently to 2.5%–3.5% of the total data size, and this low point does not increase over time.

The database maintenance tool processes the original database at the rate 7.7 – 10.4 MB/s. In our experiments, compaction reduced the database size by 30 – 50%. The exact percentage depends on the fraction of records that could be purged, which can be quite high if the file system deletes an entire snapshot line as we did in this benchmark.

6.2.2 NFS Traces

We used the first 16 days of the EECS03 trace [10], which captures research activity in home directories of a university computer science department during February and March of 2003. This is a write-rich workload, with one write for every two read operations. Thus, it places more load on Backlog than workloads with higher read-/write ratios. We ran the workload with the default configuration of 10 seconds between two consistency points.

Figure 7 shows how the overhead changes over time during the normal file system operation, omitting the cost of database maintenance. The time overhead is usually between 8 and 9 μs, which is what we saw for the synthetically generated workload, and as we saw there, the overhead remains stable over time. Unlike the overhead observed with the synthetic workload, this workload exhibits occasional spikes and one period where the overhead dips (between hours 200 and 250).

The spikes align with periods of low system load, where the constant part of the CP overhead is amortized across a smaller number of block operations, making the per-block overhead greater. We do not consider this behavior to pose any problem, since the system is under low load during these spikes and thus can better absorb the temporarily increased overhead.

The period of lower time overhead aligns with periods of high system load with a large proportion of `setattr` commands, most of which are used for file truncation. During this period, we found that only a small fraction

Benchmark	Base	Original	Backlog	Overhead
Creation of a 4 KB file (2048 ops. per CP)	0.89 ms	0.91 ms	0.96 ms	7.9%
Creation of a 64 KB file (2048 ops. per CP)	2.10 ms	2.11 ms	2.11 ms	1.9%
Deletion of a 4 KB file (2048 ops. per CP)	0.57 ms	0.59 ms	0.63 ms	11.2%
Creation of a 4 KB file (8192 ops. per CP)	0.85 ms	0.87 ms	0.87 ms	2.0%
Creation of a 64 KB file (8192 ops. per CP)	1.91 ms	1.92 ms	1.92 ms	0.6%
Deletion of a 4 KB file (8192 ops. per CP)	0.45 ms	0.46 ms	0.48 ms	7.1%
DBench CIFS workload, 4 users	19.59 MB/s	19.20 MB/s	19.19 MB/s	2.1%
FileBench /var/mail, 16 threads	852.04 ops/s	835.80 ops/s	836.70 ops/s	1.8%
PostMark	2050 ops/s	2032 ops/s	2020 ops/s	1.5%

Table 1: **Btrfs Benchmarks**. The Base column refers to a customized version of btrfs, from which we removed its original implementation of back references. The Original column corresponds to the original btrfs back references, and the Backlog column refers to our implementation. The Overhead column is the overhead of Backlog relative to the Base.

of the block operations survive past a consistency point. Thus, the operations in this interval tend to cancel each other out, resulting in smaller time overheads, because we never materialize these references in the read store.

This workload exhibits I/O overhead of approximately 0.010 to 0.015 page writes per block operation with occasional spikes, most (but not all) of which align with the periods of low file system load.

Figure 8 shows how the space overhead evolves over time for the NFS workload. The general growth pattern follows that of the synthetically generated workload with the exception that database maintenance frees less space. This is expected, since unlike the synthetic workload, the NFS trace does not delete entire snapshot lines. The space overhead after maintenance is between 6.1% and 6.3%, and it does not increase over time. The exact magnitude of the space overhead depends on the actual workload, and it is in fact different from the synthetic workload presented in Section 6.2.1. Each maintenance operation completed in less than 25 seconds, which we consider acceptable, given the elapsed time between invocations (8 or 48 hours).

6.3 Performance in btrfs

We validated our simulation results by porting our implementation of Backlog to btrfs. Since btrfs natively supports back references, we had to remove the native implementation, replacing it with our own. We present results for three btrfs configurations—the *Base* configuration with no back reference support, the *Original* configuration with native btrfs back reference support, and the *Backlog* configuration with our implementation. Comparing Backlog to the Base configuration shows the absolute overhead for our back reference implementation. Comparing Backlog to the Original configuration shows the overhead of using a general purpose back reference implementation rather than a customized implementation that is more tightly coupled to the rest of the file system.

Table 1 summarizes the benchmarks we executed on btrfs and the overheads Backlog imposes, relative to baseline btrfs. We ran microbenchmarks of create, delete, and clone operations and three application benchmarks. The create microbenchmark creates a set of 4 KB or 64 KB files in the file system's root directory. After recording the performance of the create microbenchmark, we `sync` the files to disk. Then, the delete microbenchmark deletes the files just created. We run these microbenchmarks in two different configurations. In the first, we take CPs every 2048 operations, and in the second, we take CP after 8192 operations. The choice of 8192 operations per CP is still rather conservative, considering that WAFL batches up to 32,000 operations. We also report the case with 2048 operations per CP, which corresponds to periods of a light server load as a point for comparison (and we can thus tolerate higher overheads). We executed each benchmark five times and report the average execution time (including the time to perform `sync`) divided by the total number of operations.

The first three lines in the table present microbenchmark results of creating and deleting small 4 KB files, and creating 64 KB files, taking a CP (btrfs transaction) every 256 operations. The second three lines present results for the same microbenchmarks with an inter-CP interval of 1024 operations. We show results for the three btrfs configurations—Base, Original, and Backlog. In general, the Backlog performance for writes is comparable to that of the native btrfs implementation. For 8192 operations per CP, it is marginally slower on creates than the file system with no back references (Base), but comparable to the original btrfs. Backlog is unfortunately slower on deletes – 7% as compared to Base, but only 4.3% slower than the original btrfs. Most of this overhead comes from updating the write-store.

The choice of 4 KB (one file system page) as our file size targets the worst case scenario, in which only a small number of pages are written in any given operation. The overhead decreases to as little as 0.6% for the creation of

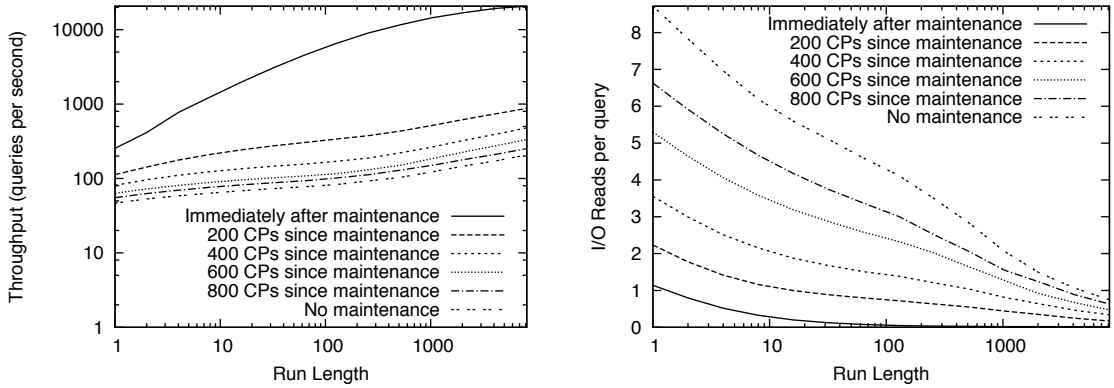


Figure 9: **Query Performance**. The query performance as a factor of run length and the number of CP's since the last maintenance on a 1000 CP-long workload. The plots show data collected from the execution of 8,192 queries with different run lengths.

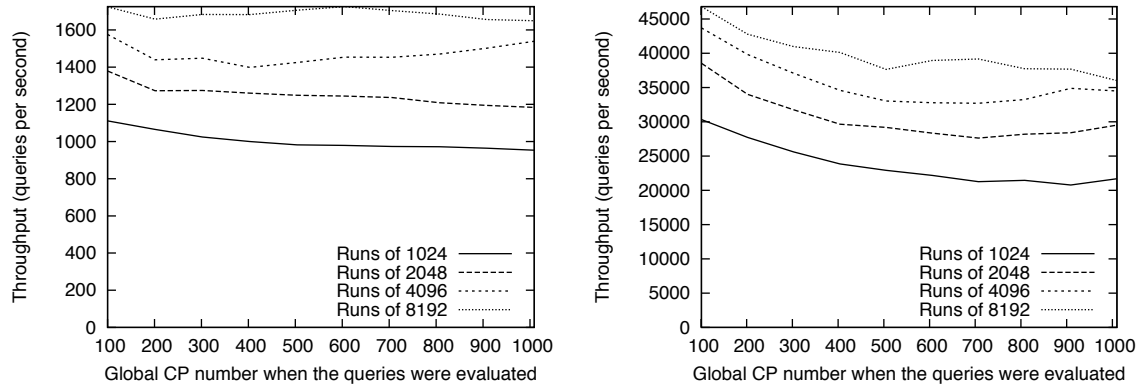


Figure 10: **Query Performance over Time**. The evolution of query performance over time on a database 100 CP's after maintenance (left) and immediately after maintenance (right). The horizontal axis is the global CP number at the time the query workload was executed. Each run of queries starts at a randomly chosen physical block.

a 64 KB file, because btrfs writes all of its data in one extent. This generates only a single back reference, and its cost is amortized over a larger number of block I/O operations.

The final three lines in Table 1 present application benchmark results: dbench [8], a CIFS file server workload; FileBench's /var/mail [11] multi-threaded mail server; and PostMark [14], a small file workload. We executed each benchmark on a clean, freshly formatted volume. The application overheads are generally lower (1.5% – 2.1%) than the worst-case microbenchmark overheads (operating on 4 KB files) and in two cases out of three comparable to the original btrfs.

Our btrfs implementation confirms the low overheads predicted via simulation and also demonstrates that Backlog achieves nearly the same performance as the btrfs native implementation. This is a powerful result as the btrfs implementation is tightly integrated with the btrfs data structures, while Backlog is a general-purpose solution that can be incorporated into any write-anywhere file system.

6.4 Query Performance

We ran an assortment of queries against the back reference database, varying two key parameters, the sequentiality of the requests (expressed as the length of a run) and the number of block operations applied to the database since the last maintenance run. We implement runs with length n by starting at a randomly selected allocated block, b, and returning back references for b and the next $n-1$ allocated blocks. This holds the amount of work in each test case constant; we always return n back references, regardless of whether the area of the file system we select is densely or sparsely allocated. It also gives us conservative results, since it always returns data for n back references. By returning the maximum possibly number of back references, we perform the maximum number of I/Os that could occur and thus report the lowest query throughput that would be observed.

We cleared both our internal caches and all file system caches before each set of queries, so the numbers we present illustrate worst-case performance. We found the

query performance in both the synthetic and NFS workloads to be similar, so we will present only the former for brevity. Figure 9 summarizes the results.

We saw the best performance, 36,000 queries per second, when performing highly sequential queries immediately after database maintenance. As the time since database maintenance increases, and as the queries become more random, performance quickly drops. We can process 290 single-back-reference queries per second immediately after maintenance, but this drops to 43 – 197 as the interval since maintenance increases. We expect queries for large sorted runs to be the norm for maintenance operations such as defragmentation, indicating that such utilities will experience the better throughput. Likewise, it is reasonable practice to run database maintenance prior to starting a query intensive task. For example, a tool that defragments a 100 MB region of a disk would issue a sorted run of at most 100 MB / 4 KB = 25,600 queries, which would execute in less than a second on a database immediately after maintenance. The query runs for smaller-scale applications, such as file defragmentation, would vary considerably – anywhere from a few blocks per run on fragmented files to thousands for the ones with a low degree of fragmentation.

Issuing queries in large sorted runs provides two benefits. It increases the probability that two consecutive queries can be satisfied from the same database page, and it reduces the total seek distance between operations. Queries on recently maintained database are more efficient for for two reasons: First, a compacted database occupies fewer RS files, so a query accesses fewer files. Second, the maintenance process shrinks the database size, producing better cache hit ratios.

Figure 10 shows the result of an experiment in which we evaluated 8192 queries every 100 CP's just before and after the database maintenance operation, also scheduled every 100 CP's. The figure shows the improvement in the query performance due to maintenance, but more importantly, it also shows that once the database size reaches a certain point, query throughput levels off, even as the database grows larger.

7 Related Work

Btrfs [2, 5] is the only file system of which we are aware that currently supports back references. Its implementation is efficient, because it is integrated with the entire file system's meta-data management. Btrfs maintains a single B-tree containing *all* meta-data objects.

A file extent back reference consists of the four fields: the subvolume, the inode, the offset, and the number of times the extent is referenced by the inode. Btrfs encapsulates all meta-data operations in transactions analogous to WAFL consistency points. Therefore a btrfs transaction ID is analogous to a WAFL CP number. Btrfs supports efficient cloning by omitting transaction ID's from back reference records, while Backlog uses ranges of snapshot versions (the `from` and `to` fields) and structural inheritance. A naïve copy-on-write of an inode in btrfs would create an exact copy of the inode (with the same inode ID), marked with a more recent transaction ID. If the back reference records contain transaction IDs (as in early btrfs designs), the file system would also have to duplicate the back references of all of the extents referenced by the inode. By omitting the transaction ID, a single back reference points to both the old and new versions of the inode simultaneously. Therefore, btrfs performs inode copy-on-write for free, in exchange for query performance degradation, since the file system has to perform additional I/O to determine transaction ID's. In contrast, Backlog enables free copy-on-write by operating on ranges of global CP numbers and by using structural inheritance, which do not sacrifice query performance.

Btrfs accumulates updates to back references in an in-memory balanced tree analogous to our write store. The system inserts all the entries from the in-memory tree to the on-disk tree during a transaction commit (a part of a checkpoint processing). Btrfs stores most back references directly inside the B-tree records that describe the allocated extents, but on some occasions, it stores them as separate items close to these extent allocation records. This is different from our approach in which we store all back references together, separately from block allocation bitmaps or records.

Perhaps the most significant difference between btrfs back references and Backlog is that the btrfs approach is deeply enmeshed in the file system design. The btrfs approach would not be possible without the existence of a global meta-store. In contrast, the only assumption necessary for our approach is the use of a write-anywhere or no-overwrite file system. Thus, our approach is easily portable to a broader class of file systems.

8 Future Work

The results presented in Section 6 provide compelling evidence that our LSM-Tree based implementation of back references is an efficient and viable approach. Our next step is to explore different options for further reducing the time overheads, the implications and effects of horizontal partitioning as described in Section 5.3, and experiment with compression. Our tables of back reference records appear to be highly compressible, especially if we to compress them by columns [1]. Compression will cost additional CPU cycles, which must be carefully balanced against the expected improvements in the space overhead.

We plan to explore the use of back references, implementing defragmentation and other functionality that uses back reference meta-data to efficiently maintain and improve the on-disk organization of data. Finally, we are currently experimenting with using Backlog in an update-in-place journaling file system.

9 Conclusion

As file systems are called upon to provide more sophisticated maintenance, back references represent an important enabling technology. They facilitate hard-to-implement features that involve block relocation, such as shrinking a partition or fast defragmentation, and enable us to do file system optimizations that involve reasoning about block ownership, such as defragmentation of files that share one or more blocks (Section 3).

We exploit several key aspects of this problem domain to provide an efficient database-style implementation of back references. By separately tracking when blocks come into use (via the `From` table) and when they are freed (via the `To` table) and exploiting the relationship between writable clones and their parents (via structural inheritance), we avoid the cost of updating per block meta-data on each snapshot or clone creation or deletion. LSM-trees provide an efficient mechanism for sequentially writing back-reference data to storage. Finally, periodic background maintenance operations amortize the cost of combining this data and removing stale entries.

In our prototype implementation we showed that we can track back-references with a low constant overhead of roughly 8-9 μs and 0.010 I/O writes per block operation and achieve query performance up to 36,000 queries per second.

10 Acknowledgments

We thank Hugo Patterson, our shepherd, and the anonymous reviewers for careful and thoughtful reviews of our paper. We also thank students of CS 261 (Fall 2009, Harvard University), many of whom reviewed our work and provided thoughtful feedback. We thank Alexei Colin for his insight and the experience of porting Backlog to other file systems. This work was made possible thanks to NetApp and its summer internship program.

References

[1] ABADI, D. J., MADDEN, S. R., AND FERREIRA, M. Integrating compression and execution in column-oriented database systems. In *SIGMOD* (2006), pp. 671–682.

[2] AURORA, V. A short history of btrfs. *LWN.net* (2009).

[3] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM 13*, 7 (July 1970), 422–426.

[4] BRODER, A., AND MITZENMACHER, M. Network applications of bloom filters: A survey. *Internet Mathematics* (2005).

[5] Btrfs. `http://btrfs.wiki.kernel.org`.

[6] CHAPMAN, A. P., JAGADISH, H. V., AND RAMANAN, P. Efficient provenance storage. In *SIGMOD* (2008), pp. 993–1006.

[7] CLEMENTS, A. T., AHMAD, I., VILAYANNUR, M., AND LI, J. Decentralized deduplication in SAN cluster file systems. In *USENIX Annual Technical Conference* (2009), pp. 101–114.

[8] DBench. `http://samba.org/ftp/tridge/dbench/`.

[9] EDWARDS, J. K., ELLARD, D., EVERHART, C., FAIR, R., HAMILTON, E., KAHN, A., KANEVSKY, A., LENTINI, J., PRAKASH, A., SMITH, K. A., AND ZAYAS, E. R. FlexVol: Flexible, efficient file volume virtualization in WAFL. In *USENIX ATC* (2008), pp. 129–142.

[10] ELLARD, D., AND SELTZER, M. New NFS Tracing Tools and Techniques for System Analysis. In *LISA* (Oct. 2003), pp. 73–85.

[11] FileBench. `http://www.solarisinternals.com/wiki/index.php/FileBench/`.

[12] HITZ, D., LAU, J., AND MALCOLM, M. A. File system design for an NFS file server appliance. In *USENIX Winter* (1994), pp. 235–246.

[13] JAGADISH, H. V., NARAYAN, P. P. S., SESHADRI, S., SUDARSHAN, S., AND KANNEGANTI, R. Incremental organization for data recording and warehousing. In *VLDB* (1997), pp. 16–25.

[14] KATCHER, J. PostMark: A new file system benchmark. *NetApp Technical Report TR3022* (1997).

[15] OLSON, M. A., BOSTIC, K., AND SELTZER, M. I. Berkeley DB. In *USENIX ATC* (June 1999).

[16] O'NEIL, P. E., CHENG, E., GAWLICK, D., AND O'NEIL, E. J. The log-structured merge-tree (LSM-Tree). *Acta Informatica 33*, 4 (1996), 351–385.

[17] QUINLAN, S., AND DORWARD, S. Venti: a new approach to archival storage. In *USENIX FAST* (2002), pp. 89–101.

[18] RODEH, O. B-trees, shadowing, and clones. *ACM Transactions on Storage 3*, 4 (2008).

[19] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10*, 1 (1992), 26–52.

[20] SCHLOSSER, S. W., SCHINDLER, J., PAPADOMANOLAKIS, S., SHAO, M., AILAMAKI, A., FALOUTSOS, C., AND GANGER, G. R. On multidimensional data and modern disks. In *USENIX FAST* (2005), pp. 225–238.

[21] SELTZER, M. I., BOSTIC, K., MCKUSICK, M. K., AND STAELIN, C. An implementation of a log-structured file system for UNIX. In *USENIX Winter* (1993), pp. 307–326.

[22] STONEBRAKER, M., ABADI, D. J., BATKIN, A., CHEN, X., CHERNIACK, M., FERREIRA, M., LAU, E., LIN, A., MADDEN, S. R., O'NEIL, E. J., O'NEIL, P. E., RASIN, A., TRAN, N., AND ZDONIK, S. B. C-Store: A column-oriented DBMS. In *VLDB* (2005), pp. 553–564.

[23] ZFS at OpenSolaris community. `http://opensolaris.org/os/community/zfs/`.

End-to-end Data Integrity for File Systems: A ZFS Case Study

Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
Computer Sciences Department, University of Wisconsin-Madison

Abstract

We present a study of the effects of disk and memory corruption on file system data integrity. Our analysis focuses on Sun's ZFS, a modern commercial offering with numerous reliability mechanisms. Through careful and thorough fault injection, we show that ZFS is robust to a wide range of disk faults. We further demonstrate that ZFS is less resilient to memory corruption, which can lead to corrupt data being returned to applications or system crashes. Our analysis reveals the importance of considering both memory and disk in the construction of truly robust file and storage systems.

1 Introduction

One of the primary challenges faced by modern file systems is the preservation of data integrity despite the presence of imperfect components in the storage stack. Disk media, firmware, controllers, and the buses and networks that connect them all can corrupt data [4, 52, 54, 58]; higher-level storage software is thus responsible for both detecting and recovering from the broad range of corruptions that can (and do [7]) occur.

File and storage systems have evolved various techniques to handle corruption. Different types of checksums can be used to detect when corruption occurs [9, 14, 49, 52], and redundancy, likely in mirrored or parity-based form [43], can be applied to recover from it. While such techniques are not foolproof [32], they clearly have made file systems more robust to disk corruptions.

Unfortunately, the effects of *memory corruption* on data integrity have been largely ignored in file system design. Hardware-based memory corruption occurs as both transient *soft errors* and repeatable *hard errors* due to a variety of radiation mechanisms [11, 35, 62], and recent studies have confirmed their presence in modern systems [34, 41, 46]. Software can also cause memory corruption; bugs can lead to "wild writes" into random memory contents [18], thus polluting memory; studies confirm the presence of software-induced memory corruptions in operating systems [1, 2, 3, 60].

The problem of memory corruption is critical for file systems that cache a great deal of data in memory for performance. Almost all modern file systems use a page cache or buffer cache to store copies of on-disk data and metadata in memory. Moreover, frequently-accessed data and important metadata may be cached in memory for long periods of time, making them more susceptible to memory corruptions.

In this paper, we ask: how robust are modern file systems to disk and memory corruptions? To answer this query, we analyze a state-of-the-art file system, Sun Microsystem's ZFS, by performing fault injection tests representative of realistic disk and memory corruptions. We choose ZFS for our analysis because it is a modern and important commercial file system with numerous robustness features, including end-to-end checksums, data replication, and transactional updates; the result, according to the designers, is "provable data integrity" [14].

In our analysis, we find that ZFS is indeed robust to a wide range of disk corruptions, thus partially confirming that many of its design goals have been met. However, we also find that ZFS often fails to maintain data integrity in the face of memory corruption. In many cases, ZFS is either unable to detect the corruption, returns bad data to the user, or simply crashes. We further find that many of these cases could be avoided with simple techniques.

The contributions of this paper are:

- To our knowledge, the first study to empirically analyze the reliability of ZFS.
- To our knowledge, the first study to analyze local file system reliability techniques in the face of memory corruption.
- A novel holistic approach to analyzing both disk and memory corruptions using carefully-controlled fault-injection techniques.
- A simple framework to measure the likelihood of different memory corruption failure scenarios.
- Results that demonstrate the importance of both memory and disk in end-to-end data protection.

The rest of this paper is organized as follows. In Section 2, we motivate our work by discussing the problem of disk and memory corruption. In Section 3, we provide some background on the reliability features of ZFS. Section 4 and Section 5 present our analysis of data integrity in ZFS with disk and memory corruptions. Section 6 gives an preliminary analysis of the probabilities of different failure scenarios in ZFS due to memory errors. In Section 7, we present initial results of the data integrity analysis in ext2 with memory corruptions. Section 8 discusses related work and Section 9 concludes our work.

2 Motivation

This section provides the motivation for our study by describing how potent the problem of disk and memory corruptions is to file system data integrity. Here, we discuss why such corruptions happen, how frequently they occur, and how systems try to deal with them. We discuss disk and memory corruptions separately.

2.1 Disk corruptions

We define disk corruption as a state when any data accessed from disk does not have the expected contents due to some problem in the storage stack. This is different from latent sector errors, not-ready-condition errors and recovered errors (discussed in [6]) in disk drives, where there is an explicit notification from the drive about the error condition.

2.1.1 Why they happen

Disk corruptions happen due to many reasons originating at different layers of the storage stack. Errors in the magnetic media lead to the problem of "bit-rot" where the magnetic properties of a single bit or few bits are damaged. Spikes in power, erratic arm movements, and scratches in media can also cause corruptions in disk blocks [4, 47, 54]. On-disk ECC catches many (but not all) of these corruptions.

Errors are also induced due to bugs in complex drive firmware (modern drives contain hundreds of thousands of lines of firmware code [44]). Some reported firmware problems include a misdirected write where the firmware accidentally writes to the wrong location [58] or a lost write (or phantom write) where the disk reports a write as completed when in fact it never reaches the disk [52]. Bus controllers have also been found to incorrectly report disk requests as complete or to corrupt data [24, 57].

Finally, software bugs in operating systems are also potential sources of corruption. Buggy device drivers can issue disk requests with bad parameters or data [20, 22, 53]. Software bugs in the file system itself can cause incorrect data to be written to disk.

2.1.2 How frequently they happen

Disk corruptions are prevalent across a broad range of modern drives. In a recent study of 1.53 million disk drives over 41 months [7], Bairavasundaram et al. show that more than 400,000 blocks had checksum mismatches, 8% of which were discovered during RAID reconstruction, creating the possibility of real data loss. They also found that nearline disks develop checksum mismatches an order of magnitude more often than enterprise class disk drives. In addition, there is much anecdotal evidence of corruption in storage stacks [9, 52, 58].

2.1.3 How to handle them

Systems use a number of techniques to handle disk corruptions. We discuss some of the most widely used techniques along with their limitations.

Checksums: Checksums are block hashes computed with a collision-resistant hash function and are used to verify data integrity. For on-disk data integrity, checksums are stored or updated on disk during write operations and read back to verify the block or sector contents during reads.

Many storage systems have used checksums for on-disk data integrity, such as Tandem NonStop [9] and NetApp Data ONTAP [52]. Similar checksumming techniques have also been used in file systems [14, 42].

However, Krioukov et al. show that checksumming, if not carefully integrated into the storage system, can fail to protect against complex failures such as lost writes and misdirected writes [32]. Further, checksumming does not protect against corruptions that happen due to bugs in software, typically in large code bases [20, 61].

Redundancy: Redundancy in on-disk structures also helps to detect and, in some cases, recover from disk corruptions. For example, some B-Tree file systems such as ReiserFS [15] store page-level information in each internal page in the B-Tree. Thus, a corrupt pointer that does not connect pages in adjacent levels is caught by checking this page-level information. Similarly, ext2 [16] and ext3 [56] use redundant copies of superblock and group descriptors to recover from corruptions.

However, it has been shown that many of these file systems still sometimes fail to detect corruptions, leading to greater problems [44]. Further, Gunawi et al. show instances where ext2/ext3 file system checkers fail to use available redundant information for recovery [26].

RAID storage: Another popular technique is to use a RAID storage system [43] underneath the file system. However, RAID is designed to tolerate the loss of a certain number of disks or blocks (e.g., RAID-5 tolerates one, and RAID-6 two) and it may not be possible with RAID alone to accurately identify the block (in a stripe) that is corrupted. Secondly, some RAID systems have been shown to have flaws where a single block loss leads to data loss or silent corruption [32]. Finally, not all systems incorporate multiple disks, which limits the applicability of RAID.

2.2 Memory corruptions

We define memory corruption as the state when the contents accessed from the main memory have one or more bits changed from the expected value (from a previous store to the location). From the software perspective, it may not be possible to distinguish memory corruption from disk corruption on a read of a disk block.

2.2.1 Why they happen

Errors in the memory chip are one source of memory corruptions. Memory errors can be classified as *soft errors* which randomly flip bits in RAM without leaving any permanent damage, and *hard errors* which corrupt bits in a repeatable manner due to physical damage.

Researchers have discovered radiation mechanisms that cause errors in semiconductor devices at terrestrial altitudes. Nearly three decades ago, May and Woods found that if an alpha particle penetrates the die surface, it can cause a random, single-bit error [35]. Zeigler and Lanford found that cosmic rays can also disrupt electronic circuits [62]. More recent studies and measurements confirm the effect of atmospheric neutrons causing single event upsets (SEU) in memories [40, 41].

Memory corruption can also happen due to software bugs. The use of unsafe languages like C and C++ makes software vulnerable to bugs such as dangling pointers, buffer overflows and heap corruption [12], which can result in seemingly random memory corruptions.

2.2.2 How frequently they happen

Early studies and measurements on memory errors provided evidence of soft errors. Data collected from a vast storehouse of data at IBM over a 15-year period [41] confirmed the presence of errors in RAM and that the upset rates increase with elevation, indicating atmospheric neutrons as the likely cause.

In a recent measurement-based study of memory errors in a large fleet of commodity servers over a period of 2.5 years [46], Schroeder et al. observe DRAM error rates that are orders of magnitude higher than previously reported, with 25,000 to 70,000 FIT per Mbit (1 FIT equals 1 failure in 10^9 device hours). They also find that more than 8% of the DIMMs they examined (from multiple vendors, with varying capacities and technologies) were affected by bit errors each year. Finally, they also provide strong evidence that memory errors are dominated by hard errors, rather than soft errors.

Another study [34] of production systems including 300 machines for a multi-month period found 2 cases of suspected soft errors and 9 cases of hard errors suggesting the commonness of hard memory faults.

Besides hardware errors, software bugs that lead to memory corruption are widely extant. Reports from the Linux Kernel Bugzilla Database [2], USCERT Vulnerabilities Notes Database [3], CERT/CC advisories [1], as well as other anecdotal evidence [18] show cases of memory corruption happening due to software bugs.

2.2.3 How to handle them

Systems use both hardware and software techniques to handle memory corruptions. Below, we discuss the most relevant hardware and software techniques.

ECC: Traditionally, memory systems have employed Error Correction Codes [19] to correct memory errors. Unfortunately, ECC is unable to address all soft-error problems. Studies found that the most commonly-used ECC algorithms called SEC/DED (Single Error Correct/Double Error Detect) can recover from only 94% of the errors in DRAMs [23]. Further, many commodity systems simply do not use ECC protection in order to reduce cost [28].

More sophisticated techniques like Chipkill[30] have been proposed to withstand multi-bit failure in DRAMs. However, such techniques are expensive and have been restricted to proprietary server systems, leaving the problem of memory corruptions open in commodity systems.

Programming models and tools: Another approach to deal with memory errors is to use recoverable programming models [38] at different levels (firmware, operating system, and applications). However, such techniques require support from hardware to detect memory corruptions. Further, a holistic change in software is required to provide recovery solution at various levels.

Much effort has also gone into detecting software bugs which cause memory corruptions. Tools such as metal [27] and CSSV [21] apply static analysis to detect memory corruptions. Others such as Purify [29] and SafeMem [45] use dynamic monitoring to detect memory corruptions at runtime. However, as discussed in Section 2.2.2, software-induced memory corruptions still remain a problem.

2.3 Summary

In modern systems corruption occurs both within the storage system and in memory. Many commercial systems apply sophisticated techniques to detect and recover from disk-level corruptions; beyond ECC, little is done to protect against memory-level problems. Therefore, the protection of critical user data against memory corruptions is largely left to software.

3 ZFS reliability features

ZFS is a state-of-the-art file system from Sun which takes a unified approach to data management. It provides data integrity, transactional consistency, scalability, and a multitude of useful features such as snapshots, copy-on-write clones, and simple administration [14].

In terms of reliability, ZFS claims to provide provable data integrity by using techniques like checksums, replication, and transactional updates. Further, the use of a pooled storage in ZFS lends it additional RAID-like reliability features. In the words of the designers, ZFS is the "The Last Word in File Systems." We now describe the reliability mechanisms in ZFS.

Checksums for data integrity checking: ZFS maintains data integrity by using checksums for on-disk

blocks. The checksums are kept separate from the corresponding blocks by storing them in the parent blocks. ZFS provides for these parental checksums of blocks by using a generic block pointer structure to address all on-disk blocks.

The block pointer structure contains the checksum of the block it references. Before using a block, ZFS calculates its checksum and verifies it against the stored checksum in the block pointer. The checksum hierarchy forms a self-validating Merkle tree [37]. With this mechanism, ZFS is able to detect silent data corruption, such as bit rot, phantom writes, and misdirected reads and writes.

Replication for data recovery: Besides using RAID techniques (described below), ZFS provides for recovery from disk corruption by keeping replicas of certain "important" on-disk blocks. Each block pointer contains pointers to up to three copies (*ditto blocks*) of the block being referenced. By default ZFS stores multiple copies for metadata and one copy for data. Upon detecting a corruption due to checksum mismatch, ZFS uses a redundant copy with a correctly-matching checksum.

COW transactions for atomic updates: ZFS maintains data consistency in the event of system crashes by using a copy-on-write transactional update model. ZFS manages all metadata and data as objects. Updates to all objects are grouped together as a transaction group. To commit a transaction group to disk, new copies are created for all the modified blocks (in a Merkle tree). The root of this tree (the *uberblock*) is updated atomically, thus maintaining an always-consistent disk image. In effect, the copy-on-write transactions along with block checksums (in a Merkle tree) preclude the need for journaling [59], though ZFS occasionally uses a write-ahead log for performance reasons.

Storage pools for additional reliability: ZFS provides additional reliability by enabling RAID-like configuration for devices using a common storage pool for all file system instances. ZFS presents physical storage to file systems in the form of a storage pool (called *zpool*). A storage pool is made up of *virtual devices* (vdev). A virtual device could be a physical device (e.g., disks) or a logical device (e.g., a mirror that is constructed by two disks). This storage pool can be used to provide additional reliability by using devices as RAID arrays. Further, ZFS also introduces a new data replication model, RAID-Z, a novel solution similar to RAID-5 but using a variable stripe width to eliminate the write-hole issue in RAID-5 [13]. Finally, ZFS provides automatic repairs in mirrored configurations and provides a disk scrubbing facility to detect latent sector errors.

4 On-disk data integrity in ZFS

In this section, we analyze the robustness of ZFS against disk corruptions. Our aim is to find whether ZFS can maintain data integrity under a variety of disk corruption scenarios. Specifically, we wish to find if ZFS can detect and recover from all disk corruptions in data and metadata and how ZFS reacts to multiple block corruptions at the same time.

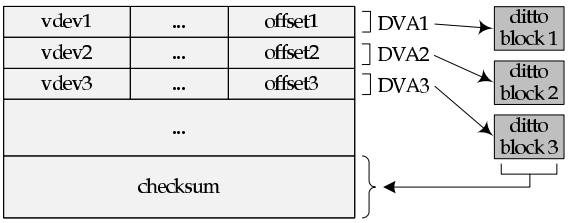


Figure 1: **Block pointer.** *The figure shows how the block pointer structure points to (up to) three copies of a block (ditto blocks), and keeps a single checksum.*

We find that ZFS is able to detect all and recover from most disk corruptions. We present our analysis, including methodology and results in later sections. First, we present a brief background about the on-disk organization in ZFS, focusing on how data integrity is maintained.

4.1 ZFS on-disk organization

All on-disk data and metadata in ZFS are treated as objects, where an object is a collection of blocks. Objects are further grouped into object sets. Other structures such as uberblocks are also used to organize data on disk. We now discuss these basic on-disk structures and their usage in ZFS.

4.1.1 Basic structures

Block pointers: A block pointer is the basic structure in ZFS for addressing a block on disk. It provides a generic mechanism to keep parental checksums and replicas of on-disk blocks. Figure 1 shows the block pointer used by ZFS. As shown, the block pointer contains up to three block addresses, called DVAs (*data virtual addresses*), each pointing to a different block having the same contents. These are referred to as *ditto blocks*. The number of DVAs varies depending on the importance of the block. The current policy in ZFS is that there is one DVA for user data, two DVAs for file system metadata, and three DVAs for global metadata across all file system instances in the pool [39]. As discussed earlier, the block pointer also contains a single copy of the checksum of the block being pointed to.

Objects: All blocks on disk are organized in objects. Physically, an object is represented on disk by a structure called `dnode_phys_t` (hereafter referred to as *dnode*). A dnode contains an array of up to three block pointers, each of which points to either a leaf block (e.g., a data block) or an indirect block (full of block pointers). These blocks pointed to by the dnode form a block tree. A dnode also contains a bonus buffer at the end, which stores an object-specific data structure for different types

Level	Object Name	Simplified Explanation
zpool	MOS dnode	A dnode object that contains dnode blocks, which store dnodes representing pool-level objects.
	Object directory	A ZAP object whose blocks contain name-value pairs referencing further objects in the MOS object set.
	Dataset	It represents an object set (e.g., a file system) and tracks its relationships with its snapshots and clones.
	Dataset directory	It maintains an active dataset object along with its child datasets. It has a reference to a dataset child map object. It also maintains properties such as quotas for all datasets in this directory.
	Dataset child map	A ZAP object whose blocks hold name-value pairs referencing child dataset directories.
zfs	FS dnode	A dnode object that contains dnode blocks, which store dnodes representing filesystem-level objects.
	Master node	A ZAP object whose blocks contain name-value pairs referencing further objects in this file system.
	File	An object whose blocks contain file data.
	Directory	A ZAP object whose blocks contain name-value pairs referencing files and directories inside this directory.

Table 1: **Summary of ZFS objects visited.** *The table presents a summary of all ZFS objects visited in the walkthrough, along with a simplified explanation. Note that ZAP stands for ZFS Attribute Processor. A ZAP object is used to store name-value pairs.*

of objects. For example, a dnode of a file object contains a structure called `znode_phys_t` (*znode*) in the bonus buffer, which stores file attributes such as access time, file mode and size of the file.

Object sets: Object sets are used in ZFS to group related objects. An example of a object set is a file system, which contains file objects and directory objects belonging to this file system.

An object set is represented by a structure called `objset_phys_t`, which consists of a meta dnode and a ZIL (ZFS Intent Log) header. The meta dnode points to a group of dnode blocks; dnodes representing the objects in this object set are stored in these dnode blocks. The object described by the meta dnode is called "dnode object". The ZIL header points to a list of blocks, which holds transaction records for ZFS's logging mechanism.

Other structures: ZFS uses other structures to organize on-disk data. Each physical vdev is labeled with a *vdev label* that describes this device and other related virtual devices. Four copies of the label are stored in each physical vdev to provide redundancy and a two-stage update mechanism is used to guarantee that there is always a valid vdev label in the device [51]. An *uberblock* (similar to a superblock) inside the vdev label is used to provide access to the pool data and verify its integrity. The uberblock is self-checksummed and updated atomically.

4.1.2 On-disk layout

In this section, we present some details about ZFS on-disk layout. This overview will help the reader to understand the range of our fault injection experiments presented in later sections. A complete description of ZFS on-disk structures can be found elsewhere [51].

For the purpose of illustration, we demonstrate the steps that ZFS takes to locate a file system and to locate file data in it in a simple storage pool. Figure 2 shows the on-disk layout of the simplified pool with a sample file system called "myfs", along with the sequence of objects and blocks accessed by ZFS. A simple explanation of all visited objects is described in Table 1. Note that we skip the details of how in-memory structures are set up and assume that data and metadata are not cached in memory to begin with.

Find pool metadata (steps 1-2): As the starting point, ZFS locates the active uberblock in the vdev label of the device. ZFS then uses the uberblock to locate and verify the integrity of pool-wide metadata contained in an object set called Meta Object Set (MOS). There are three copies of the object set block representing the MOS.

Find a file system (steps 3-10): To locate a file system, ZFS accesses a series of objects in MOS, all of which have three ditto blocks. Once the dataset representing "myfs" is found, it is used to access file system wide metadata contained in an object set. The integrity of file system metadata is checked using the block pointer in the dataset, which points to the object set block. All file system metadata blocks have two ditto copies.

Find a file and a data block (steps 11-18): To locate a file, ZFS then uses the directory objects in the "myfs" object set. Finally, by following the block pointers in the dnode of the file object, ZFS finds the required data block. The integrity of every traversed block is confirmed by verifying the checksum in its block pointers.

The legend in Figure 2 shows a summary of all the on-disk block types visited during the traversal. Our fault injection tests for analyzing robustness of ZFS against disk corruptions (discussed in the next subsection) inject bit errors in the on-disk blocks shown in Figure 2.

4.2 Methodology of analysis

In this section, we discuss the methodology of our reliability analysis of ZFS against disk corruptions. We discuss our fault injection framework first and then present our test procedures and workloads.

4.2.1 Fault injection framework

Our experiments are performed on a 64-bit Solaris Express Community Edition (build 108) virtual machine with 2GB non-ECC memory. We use ZFS pool version 14 and ZFS filesystem version 3. We run ZFS on top of a single disk for our experiments.

To emulate disk corruptions, we developed a fault injection framework consisting of a pseudo-driver to perform fault injection on disk blocks and an application for

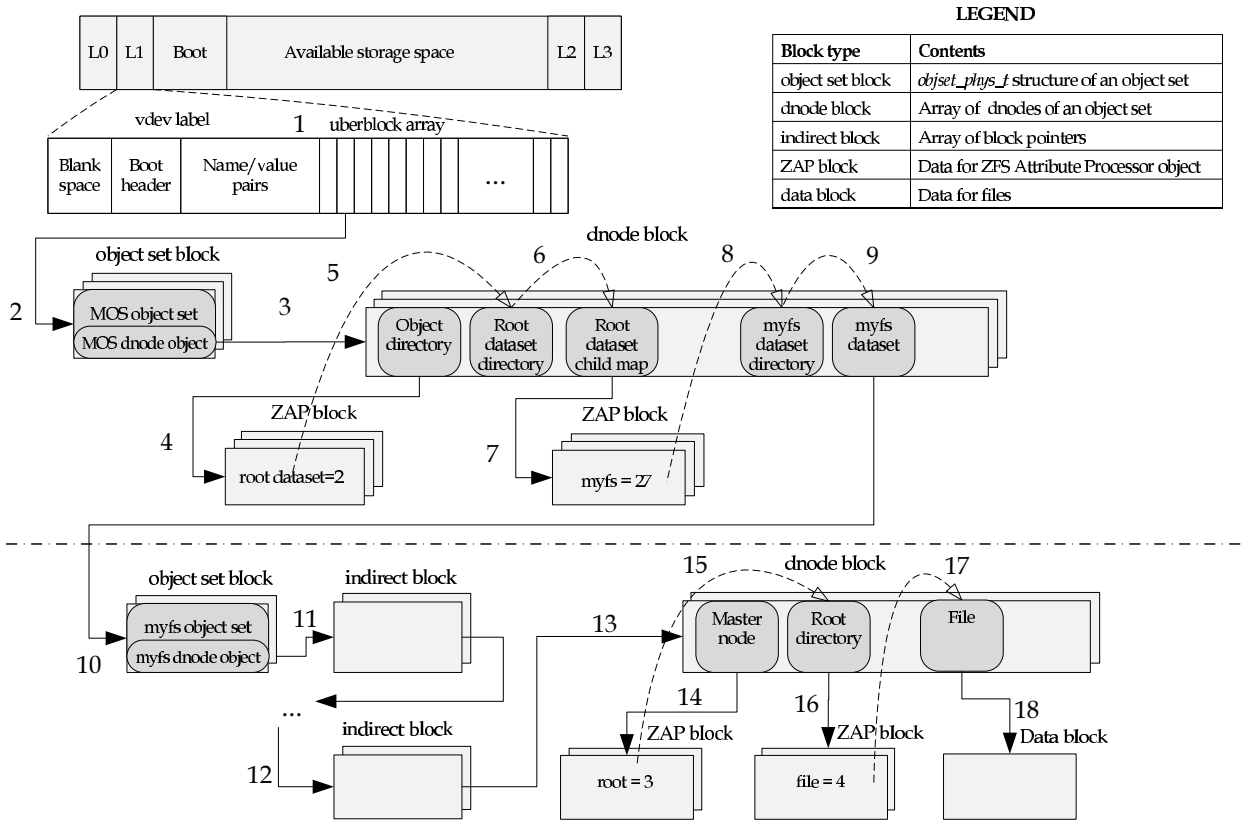


Figure 2: **ZFS on-disk structures.** *The figure shows the on-disk structures of ZFS including the pool-wide metadata and file system metadata. In the example above, the zpool contains a sample file system named "myfs". All ZFS on-disk data structures are shown by rounded boxes, and on-disk blocks are shown by rectangular boxes. Solid arrows point to allocated blocks and dotted arrows represent references to objects inside blocks. The legend at the top shows the types of on-disk blocks and their contents.*

controlling the experiments. The pseudo-driver is a standard Solaris layered driver that interposes between the ZFS virtual device and the disk driver beneath. We analyze the behavior of ZFS by looking at return values, checking system logs, and tracing system calls.

4.2.2 Test procedure and workloads

In our tests, we wanted to understand the behavior of ZFS to disk corruptions on different types of blocks. We injected faults by flipping bits at random offsets in disk blocks. Since we used the default setting in ZFS for compression (metadata compressed and data uncompressed), our fault injection tests corrupted compressed metadata and uncompressed data blocks on disk. We injected faults on nine different classes of ZFS on-disk blocks and for each class, we corrupted a single copy as well as all copies of blocks.

In our fault injection experiments on pool-wide and file system level metadata, we used "mount" and "remount" operations as our workload. The "mount" workload indicates that the target block is corrupted with the pool exported and "myfs" not mounted, and we subsequently mount it. This workload forces ZFS to use on-disk copies of metadata. The "remount" workload indicates that the target block is corrupted with "myfs" mounted and we subsequently umount and mount it. ZFS uses in-memory copies of metadata in this workload.

For injecting faults in file and directory blocks in a file system, we used two simple operations as workloads: "create file" creates a new file in a directory, and "read file" reads a file's contents.

4.3 Results and observations

The results of our fault injection experiments are shown in Table 2. The table reports the results of experiments on pool-wide metadata and file system metadata and data. It also shows the results of corrupting a single copy as well as all copies of blocks. We now explain the results in detail in terms of the observations we made from our fault injection experiments.

Observation 1: *ZFS detects all corruptions due to the use of checksums.* In our fault injection experiments on all metadata and data, we found that bad data was never returned to the user because ZFS was able to detect all corruptions due to the use of checksums in block pointers. The parental checksums are used in ZFS to ver-

Level	Block	Single ditto: mount	Single ditto: remount	Single ditto: create file	Single ditto: read file	All ditto: mount	All ditto: remount	All ditto: create file	All ditto: read file
zpool	vdev label[1]	R	R			E	R		
	uberblock	R	R			E	R		
	MOS object set block	R	R			E	R		
	MOS dnode block	R	R			E	R		
zfs	myfs object set block	R	R			E	R		
	myfs indirect block	R	R			E	R		
	myfs dnode block	R	R			E	R		
	dir ZAP block			R	R			E	E
	file data block				E				E

[1] excluding the uberblocks contained in it.

Table 2: **On-disk corruption analysis.** *The table shows the results of on-disk experiments. Each cell indicates whether ZFS was able to recover from the corruption (R), whether ZFS reported an error (E), whether ZFS returned bad data to the user (B), or whether the system crashed (C). Blank cells mean that the workload was not exercised for the block.*

ify the integrity of all the on-disk blocks accessed. The only exception are uberblocks, which do not have parent block pointers. Corruptions to the uberblock are detected by the use of checksums inside the uberblock itself.

Observation 2: *ZFS gracefully recovers from single metadata block corruptions*. For pool-wide metadata and file system wide metadata, ZFS recovered from disk corruptions by using the ditto blocks. ZFS keeps three ditto blocks for pool-wide metadata and two for file system metadata. Hence, on single-block corruption to metadata, ZFS was successfully able to detect the corruption and use other available correct copies to recover from it; this is shown by the cells (R) in the "Single ditto" column for all metadata blocks.

Observation 3: *ZFS does not recover from data block corruptions*. For data blocks belonging to files, ZFS was not able to recover from corruptions. ZFS detected the corruption and reported an error on reading the data block. Since ZFS does not keep multiple copies of data blocks by default, this behavior is expected; this is shown by the cells (E) for the file data block.

Observation 4: *In-memory copies of metadata help ZFS to recover from serious multiple block corruptions*. In an active storage pool, ZFS caches metadata in memory for performance. ZFS performs operations on these cached copies of metadata and writes them to disk on transaction group commits. These in-memory copies of metadata, along with periodic transaction commits, help ZFS recover from multiple disk corruptions.

In the "remount" workload that corrupted all copies of uberblock, ZFS recovered from the corruptions because the in-memory copy of the active uberblock remains as long as the pool exists. The in-memory copy is subsequently written to a new disk block in a transaction group commit, making the old corrupted copy void. Similar results were obtained when corrupting other pool-wide metadata and file system metadata, and ZFS was able to recover from these multiple block corruptions (R).

Observation 5: *ZFS cannot recover from multiple block corruptions affecting all ditto blocks when no in-memory copy exists*. For file system metadata, like directory ZAP blocks, ZFS does not always keep an in-memory copy unless the directory has been accessed. Thus, on corruptions to both ditto blocks, ZFS reported an error. This behavior is shown by the results (E) for directories indicating for the "create file" and "read file" operations. Note that we performed these corruptions without first accessing the directory, so that there were no in-memory copies. Similarly, in the "mount" workload, when the pool was inactive (exported) and thus no in-memory copies existed, ZFS was unable to recover from multiple disk corruptions and responded with errors (E).

Observation 4 and 5 also lead to an interesting conclusion that an active storage pool is likely to tolerate more serious disk corruptions than an inactive one.

In summary, ZFS successfully detects all corruptions and recovers from them as long as one correct copy exists. The in-memory caching and periodic flushing of metadata on transaction commits help ZFS recover from serious disk corruptions affecting all copies of metadata. For user data, ZFS does not keep redundant copies and is unable to recover from corruptions. ZFS, however, detects the corruptions and reports an error to the user.

5 In-memory data integrity in ZFS

In the last section we showed the robustness of ZFS to disk corruptions. Although ZFS was not specifically designed to tolerate memory corruptions, we still would like to know how ZFS reacts to memory corruptions, i.e., whether ZFS can detect and recover from a single bit flip in data and metadata blocks. Our fault injection experiments indicate that ZFS has no precautions for memory corruptions: bad data blocks are returned to the user or written to disk, file system operations fail, and many times the whole system crashes.

This section is organized as follows. First, we briefly describe ZFS in-memory structures. Then, we discuss the test methodology and workloads we used to conduct the analysis. Finally, we present the experimental results and our observations.

5.1 ZFS in-memory structures

In order to better understand the in-memory experiments, we present some background information on ZFS in-memory structures.

5.1.1 In-memory structures

ZFS in-memory structures can be classified into two categories: those that exist in the page cache and those that

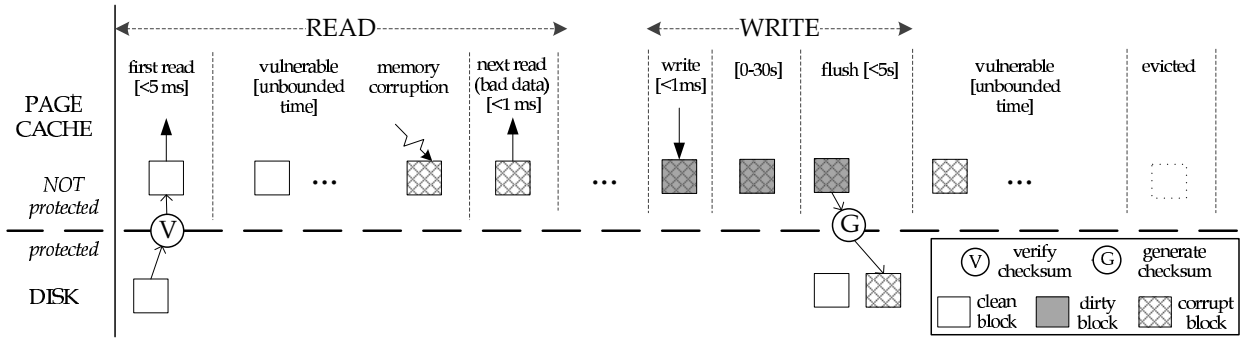


Figure 3: **Lifecycle of a block.** *This figure illustrates one example of the lifecycle of a block. The left half represents the read timeline and the right half represents the write timeline. The black dotted line is a protection boundary, below which a block is protected by the checksum, otherwise unprotected.*

are in memory outside of the page cache; for convenience we call the latter *in-heap* structures. Whenever a disk block is accessed, it is loaded into memory. Disk blocks containing data and metadata are cached in the ARC page cache [36], and stay there until evicted. Data blocks are stored only in the page cache, while most metadata structures are stored in both the page cache (as copies of on-disk structures) and the heap. Note that block pointers inside indirect blocks are also metadata, but they only reside in the page cache. Uberblocks and vdev labels, on the other hand, only stay in the heap.

5.1.2 Lifecycle of a block

To help the reader understand the vulnerability of ZFS to memory corruptions discussed in later sections, Figure 3 illustrates one example of the lifecycle of a block (i.e., how a block is read from and written asynchronously to disk). To simplify the explanation, we consider a pair of blocks in which the target block to be read or written is pointed to by a block pointer contained in the parental block. The target block could be a data block or a metadata block. The parental block could be an indirect block (full of block pointers), a dnode block (array of dnodes, each of which contains block pointers) or an object set block (a dnode is embedded in it). The user of the block could be a user-level application or ZFS itself. Note that only the target block is shown in the figure.

At first, the target block is read from disk to memory. For read, there are two scenarios, as shown in the left half of Figure 3. On first read of a target block not in the page cache, it is read from the disk and immediately verified against the checksum stored in the block pointer in the parental block. Then the target block is returned to the user. On a subsequent read of a block already in the page cache, the read request gets the cached block from the page cache directly, without verifying the checksum.

In both cases, after the read, the target block stays in the page cache until evicted. The block remains in the page cache for an unbounded interval of time depending on many factors such as the workload and the cache replacement policy.

After some time, the block is updated. The write timeline is illustrated in the right half of Figure 3. All updates are first done in the page cache and then flushed to disk. Thus before the updates occur, the target block is either in the page cache already or just loaded to the page cache from disk. After the write, the updated block stays in the page cache for at most 30 seconds and then it is flushed to disk. During the flush, a new physical block is allocated and a new checksum is generated for the dirty target block. The new disk address and checksum are then written to the block pointer contained in the parental block, thus making it dirty. After the target block is written to the disk, the flush procedure continues to allocate a new block and calculate a new checksum for the parental block, which in turn dirties its subsequent parental block. Following the updates of block pointers along the tree (solid arrows in Figure 2), it finally reaches the uberblock which is self-checksummed. After the flush, the target block is kept in the page cache until it is evicted.

5.2 Methodology of analysis

In this section, we discuss the fault injection framework, and the test procedure and workloads. The injection framework is similar to the one used for on-disk experiments. The only difference is the pseudo-driver, which in this case, interacts with the ZFS stack by calling internal functions to locate the in-memory structures.

5.2.1 Test procedure and workloads

We wished to find out the behavior of ZFS in response to corruptions in different in-memory objects. Since all data and metadata in memory are uncompressed, we performed a controlled fault injection in various objects. For metadata, we randomly flipped a bit in each individual field of the structure separately; for data, we randomly corrupted a bit in a data block of a file in memory. We repeated each fault injection test five times. We performed

Object	Data Structures	Workload
MOS dnode	dnode_t, dnode_phys_t	zfs create, zfs destroy, zfs rename, zfs list, zfs mount, zfs umount
Object directory	dnode_t, dnode_phys_t, mzap_phys_t, mzap_ent_phys_t	
Dataset	dnode_t, dnode_phys_t, dsl_dataset_phys_t	
Dataset directory	dnode_t, dnode_phys_t, dsl_dir_phys_t	
Dataset child map	dnode_t, dnode_phys_t, mzap_phys_t, mzap_ent_phys_t	
FS dnode	dnode_t, dnode_phys_t	zfs umount, path traversal
Master node	dnode_t, dnode_phys_t, mzap_phys_t, mzap_ent_phys_t	
File	dnode_t, dnode_phys_t, znode_phys_t	open, close, lseek, read, write, access, link, unlink, rename, truncate (chdir, mkdir, rmdir)
Dir	dnode_t, dnode_phys_t, znode_phys_t, mzap_phys_t, mzap_ent_phys_t	

Table 3: **Summary of objects and data structures corrupted.** *The table presents a summary of all the ZFS objects and structures corrupted in our in-memory analysis, along with their data structures and the workloads exercised on them.*

Data Structure	Fields
dnode_t	dn_nlevels, dn_bonustype, dn_indblkshift, dn_nblkptr, dn_datablkszsec, dn_maxblkid, dn_compress, dn_bonuslen, dn_checksum, dn_type
dnode_phys_t	dn_nlevels, dn_bonustype, dn_indblkshift, dn_nblkptr, dn_datablkszsec, dn_maxblkid, dn_compress, dn_bonuslen, dn_checksum, dn_type, dn_used, dn_flags,
mzap_phys_t	mz_block_type, mz_salt
mzap_ent_phys_t	mze_value, mze_name
znode_phys_t	zp_mode, zp_size, zp_links, zp_flags, zp_parent
dsl_dir_phys_t	dd_head_dataset_obj, dd_child_dir_zapobj, dd_parent_obj
dsl_dataset_phys_t	ds_dir_obj

Table 4: **Summary of data structures and fields corrupted.** *The table lists all fields we corrupted in the in-memory experiments.* `mzap_phys_t` *and* `mzap_ent_phys_t` *are metadata stored in ZAP blocks. The last three structures are object-specific structures stored in the dnode bonus buffer.*

fault injection tests on nine different types of objects at two levels (zfs and zpool) and exercised different set of workloads as listed in Table 3. Table 4 shows all data structures inside the objects and all the fields we corrupted during the experiments.

For data blocks, we injected bit flips at an appropriate time as described below. For reads, we flipped a random bit in the data block after it was loaded to the page cache; then we issued a subsequent read() on that block to see if ZFS returned the corrupted block. In this case, the read() call fetched the block from the page cache. For writes, we corrupted the block after the write() call finished but before the target block was written to the disk.

For metadata, in our fault injection experiments, we covered a broad range of metadata structures. However, to reduce the sample space for experiments to more interesting cases, we made two choices. First, we always injected faults to the in-memory structure after it was accessed by the file system, so that both the in-heap version and page cache version already exist in the memory. Second, among the in-heap structures, we only corrupted the `dnode_t` structure (in-heap version of `dnode_phys_t`). The dnode structure is the most widely used metadata structure in ZFS and every object in ZFS is represented by a dnode. Hence, we anticipate that corrupting the in-heap dnode structure will cover many interesting cases.

5.3 Results and observations

We present the results of our in-memory experiments in Table 5. As shown, ZFS fails to catch data block corruptions due to memory errors in both read and write experiments. Single bit flips in metadata blocks not only lead to returning bad data blocks, but also cause more serious problems like failure of operations and system crashes. Note that Table 5 is a subset of the results showing only cases with apparent problems. In other cases that are either indicated by a dot (.) in the result cells or not shown at all in Table 5, the corresponding operation either did not access the corrupted field or completed successfully with the corrupted field. However, in all cases, ZFS did not correct the corrupted field.

Next we present our observations on ZFS behavior and user-visible results. The first five observations are about ZFS behavior and the last five observations are about user-visible results of memory corruptions.

Observation 1: *ZFS does not use the checksums in the page cache along with the blocks to detect memory corruptions.* Checksums are the first guard for detecting data corruption in ZFS. However, when a block is already in the page cache, ZFS implicitly assumes that it is protected against corruptions. In the case of reads, the checksum is verified only when the block is being read from the disk. Following that, as long as the block stays in the page cache, it is never checked against the checksum, despite the checksum also being in the page cache (in the block pointer contained in its parental block). The result is that ZFS returns bad data to the user on reads.

For writes, the checksum is generated only when the block is being written to disk. Before that, the dirty block stays in the page cache with an outdated checksum in the block pointer pointing to it. If the block is corrupted in the page cache before it is flushed to disk, ZFS calculates a checksum for the bad block and stores the new checksum in the block pointer. Both the block and its parental block containing the block pointer are written to disk. On subsequent reads of the block, it passes the checksum verification and is returned to the user.

Moreover, since the detection mechanisms already fail to detect memory corruptions, recovery mechanisms

		File	Dir	MOS dnode	Dataset directory	Dataset childmap	Dataset
Structure	**Field**	**O R W A U N T**	**O A L U N T M C D**	**c d r l m u**	**c d r l m u**	**c d r**	**c d r l m**
dnode_t	dn_type	. . . . .	. . . . .	C C C C C C	. . . . .	. . .	. . . . .
	dn_indblkshift	. BC . . C . .	. . E E E . E . E	. . . . .	. . . . .	. . .	. . . . .
	dn_nlevels	. . C . . . C	. . C C C . C . C	C C C C C C	. . . . .	C C C	C C C . .
	dn_checksum	. . C . . . C	. . . . .	. . . . .	. . . . .	. . .	. . . . .
	dn_compress	. . C	. . . . .	. . . . .	. . . . .	. . .	. . . . .
	dn_maxblkid	 C	 C	. . . . .	. . . . .	. . .	. . . . .
dnode_phys_t	dn_indblkshift	. . . . C . .	. . . . .	. . . . .	. . . . .	. . .	. . . . .
	dn_nlevels	. BC C . C . .	 C	. . . . .	. . . . .	. C .	. . . . .
	dn_nblkptr	. . . . .	. . . . .	. . . . .	. . . . .	. . .	. C . . .
	dn_bonuslen	. . C	. . . . .	. . . . .	. C . . .	. . .	. C . . .
	dn_maxblkid	. B . . C . C	 C	. . . . .	. C . . .	. C .	. C . . .
znode_phys_t	zp_size	. . . . .	. . . . . E				
	zp_flags	E . . E . E E	E E E E E E E E				
dsl_dir_phys_t	dd_head_dataset_obj				E E E E . .		
	dd_child_dir_zapobj				EC EC EC EC EC C		
dsl_dataset_phys_t	ds_dir_obj						. E E . .
data block		B B					

Table 5: **In-memory corruption results.** *The table shows a subset of memory corruption results. The operations exercised are O(open), R(read), W(write), A(access), L(link), U(unlink), N(rename), T(truncate), M(mkdir), C(chdir), D(rmdir), c(zfs create), d(zfs destroy), r(zfs rename), l(zfs list), m(zfs mount) and u(zfs umount). Each result cell indicates whether the system crashed (C), whether the operation failed with wrong results or with a misleading message (E), whether a bad data block was returned (B) or whether the operation completed (.). Large blanks mean that the operations are not applicable.*

such as ditto blocks and the mirrored zpool are not triggered to recover from the damage.

The results in Table 5 indicate that when a data block was corrupted, the application that issued a read() or write() request was returned bad data (B), as shown in the last row. When metadata blocks were corrupted, ZFS accessed the corrupted data structures and thus behaved wrongly, as shown by other cases in the result table.

Observation 2: *The window of vulnerability of blocks in the page cache is unbounded.* As Figure 3 shows, after a block is loaded into the page cache by first read, it stays there until evicted. During this interval, if a corruption happens to the block, any subsequent read will get the corrupted block because the checksum is not verified. Therefore, as long as the block is in the page cache (unbounded), it is susceptible to memory corruptions.

Observation 3: *Since checksums are created when blocks are written to disk, any corruption to blocks that are dirty (or will be dirtied) is written to disk permanently on a flush*. As described in Section 5.1.2, dirty blocks in the page cache are written to disk during a flush. During the flush, any dirty block will further cause updates of all its parental blocks; a new checksum is then calculated for each updated block and all of them are flushed to disk. If a memory corruption happens to any of those blocks before a flush (above the black dotted line before G in Figure 3), the corrupted block is written to disk with a new checksum. The checksum is thus valid for the corrupted block, which makes the corruption permanent. Since the window of vulnerability is long (30 seconds), and there are many blocks that will be flushed to disk in each flush, we conjecture that the likelihood of memory corruption leading to permanent on-disk corruptions is high.

We did a block-based fault injection to verify this observation. We injected a single bit flip to a dirty (or to-be-dirtied) block before a flush; as long as the flipped bit in the block was not overwritten by subsequent operations, the corrupted block was written to disk permanently.

Observation 4: *Dirtying blocks due to updating file access time increases the possibility of making corruptions permanent*. By default, access time updates are enabled in ZFS; therefore, a read-only workload will update the access time of any file accessed. Consequently, when the structure containing the access time (znode) goes inactive (or when there is another workload that updates the znode), ZFS writes the block holding the znode to disk and updates and writes all its parental blocks. Therefore, any corruption to these blocks will become permanent after the flush caused by the access time update. Further, as mentioned earlier, the time interval when the corruption could happen is unbounded.

Observation 5: *For most metadata blocks in the page cache, checksums are not valid and thus useless in detecting memory corruptions.* By default, most metadata blocks such as indirect blocks and dnode blocks are compressed on disk. Since the checksums for these blocks are used to prevent disk corruptions, they are only valid for compressed blocks, which are calculated after they are compressed during writes and verified before they are decompressed during reads. When metadata blocks are in the page cache, they are uncompressed. Therefore, the

checksums contained in the corresponding block pointers are useless.

We now discuss our observations about user-visible results of memory corruptions.

Observation 6: *When metadata is corrupted, operations fail with wrong results, or give misleading error messages (E).* As shown in Table 5, when `zp_flags` in `dnode_phys_t` for a file object was corrupted, in one case open() returned an error code EACCES (permission denied). This case occurred when the 41st bit of `zp_flags` was flipped from 0 to 1, which signifies that the file is quarantined by an anti-virus software. Therefore, open() was incorrectly denied, giving an error code EACCES. The calls access(), rename() and truncate() also failed for the same reason.

Another example of a misleading error message happened when `dd_head_dataset_obj` in `dsl_dir_phys_t` for a dataset directory object was corrupted; there is one case where "zfs create" failed to create a new file system under the parent file system represented by the corrupted object. ZFS gave a misleading error message saying that the parent file system did not exist. ZFS gave similar error messages in other cases (E) under "Dataset directory" and "Dataset".

A case where wrong results are returned occurred when `dd_child_dir_zapobj` was corrupted. This field refers to a dataset child map object containing references to child file systems. On corrupting this field, "zfs list", which should list all file systems in the pool, did not list the child file systems of the corrupted dataset directory.

Observation 7: *Many corruptions lead to a system crash (C).* For example, when `dn_nlevels` (the height of the block tree pointed to by the dnode) in `dnode_phys_t` for a file object was corrupted and the file was read, the system crashed due to a NULL pointer dereference. In this case, ZFS used the wrong value of `dn_nlevels` to traverse the block tree of the file object and obtained an invalid block pointer. Therefore, the block size obtained from the block pointer was an arbitrary value, which was then used to index into an array whose size was much less than the value. As a result, the system crashed when a NULL pointer was dereferenced.

Observation 8: *The read() system call may return bad data.* As shown in Table 5, for metadata corruptions, there were three cases where read() gave bad data block to the user. In these cases, ZFS simply trusted the value of the corrupted field and used it to traverse the block tree pointed to by the dnode, thus returning bad blocks. For example, when `dn_nlevels` in `dnode_phys_t` for a file object was changed from 3 to 1, ZFS gave an incorrect block to the user on a read request for the first block of the file. The bad block was returned because ZFS assumed that the tree only had one level, and incorrectly returned an indirect block to the user. Such cases where wrong blocks are returned to the user also have the potential for security vulnerabilities.

Observation 9: *There is no recovery for corrupted metadata.* In the cases where no apparent error happened (as indicated by a dot or not shown) and the operation was not meant to update the corrupted field, the corruption remained in the metadata block in the page cache.

In summary, ZFS fails to detect and recover from many corruptions. Checksums in the page cache are not used to protect the integrity of blocks. Therefore, bad data blocks are returned to the user or written to disk. Moreover, corrupted metadata blocks are accessed by ZFS and lead to operation failure and system crashes.

6 Probability of bit-flip induced failures

In this section, we present a preliminary analysis of the likelihood of different failure scenarios due to memory errors in a system using ZFS. Specifically, given that one random bit in memory is flipped, we compute the probabilities of four scenarios: reading corrupt data (R), writing corrupt data (W), crashing/hanging (C) and running successfully to complete (S). These probabilities help us to understand how severely filesystem data integrity is affected by memory corruptions and how much effort filesystem developers should make to add extra protection to maintain data integrity.

6.1 Methodology

We apply fault-injection techniques to perform the analysis. Considering one run of a specific workload as a trial, we inject a fixed number number of random bit flips to the memory and record how the system reacts. Therefore, by doing multiple trials, we measure the number of trials where each scenario occurs, thus estimating the probability of each scenario given that certain number of bits are flipped. Then, we calculate the probability of each scenario given the occurrence of one single bit flip.

We have extended our fault injection framework to conduct the experiments. We replaced the pseudo-driver with a user-level "injector" which injects random bit flips to the physical memory. We used filebench [50] to generate complex workloads. We modified filebench such that it always writes predefined data blocks (e.g., full of 1s) to disk. Therefore, we can check every read operation to verify that the returned data matches the predefined pattern. We can also verify the data written to disk by checking the contents of on-disk files.

We used the framework as follows. For a specific workload, we ran 100 trials. For each trial, we used the injector to generate 16 random bit flips at the same time when the workload has been running for 3 minutes. We then kept the workload running for 5 minutes. Any occurrence of reading corrupt data (R) was reported. When the workload was done, we checked all on-disk files to

see if there was any corrupt data written to the disk (W). Since we only verify write operations after each run of a workload, some intermediate corrupt data might have been overwritten and thus the actual number of occurrence of writing corrupt data could be higher than measured here. We also logged whether the system hung or crashed (C) during each trial, but we did not determine if it was due to corruption of ZFS metadata or other kernel data structures.

It is important to notice that we injected 16 bit flips in each trial because it let us observe a sufficient number of failure trials in 100 trials. However, we apply the following calculation to derive the probabilities of different failure scenarios given that 1 bit is flipped.

6.2 Calculation

We use $P_k(X)$ to represent the probability of scenario X given that k random bits are flipped, in which X could be R, W, C or S. Therefore, $P_k(\bar{X}) = 1 - P_k(X)$ is the probability of scenario X not happening given that k bits are flipped. In order to calculate $P_1(X)$, we first measure $P_k(X)$ using the method described above and then derive $P_1(X)$ from $P_k(X)$, as explained below.

- **Measure** $P_k(X)$ Given that k random bit flips are injected in each trial, we denote the total number of trials as N and the number of trials in which scenario X occurs at least once as N_X. Therefore,
$$P_k(X) = \frac{N_X}{N}$$
- **Derive** $P_1(X)$ Assume k bit flips are independent, then we have
$$P_k(\bar{X}) = (P_1(\bar{X}))^k, \text{ when } X = R, W \text{ or } C$$
$$P_k(X) = (P_1(X))^k, \text{ when } X = S$$
Substituting $P_k(\bar{X}) = 1 - P_k(X)$ into the equations above, we can get,
$$P_1(X) = 1 - (1 - P_k(X))^{\frac{1}{k}}, \text{ when } X = R, W \text{ or } C$$
$$P_1(X) = (P_k(X))^{\frac{1}{k}}, \text{ when } X = S$$

6.3 Results

The analysis is performed on the same virtual machine as mentioned in Section 4.2.1. The machine is configured with 2GB non-ECC memory and a single disk running ZFS. We first ran some controlled micro-benchmarks (e.g., sequential read) to verify that the methodology and the calculation is correct (the result is not shown due to limited space). Then, we chose four workloads from filebench: varmail, oltp, webserver and fileserver, all of which were exercised with their default parameters. A detailed description of these workloads can be found elsewhere [50].

Workload	$P_{16}(R)$	$P_{16}(W)$	$P_{16}(C)$	$P_{16}(S)$
varmail	9% [4, 17]	0% [0, 3]	5% [1, 12]	86% [77, 93]
oltp	26% [17, 36]	2% [0, 8]	16% [9, 25]	60% [49, 70]
webserver	11% [5, 19]	20% [12, 30]	19% [11, 29]	61% [50, 71]
fileserver	69% [58, 78]	44% [34, 55]	23% [15, 33]	28% [19, 38]

Workload	$P_1(R)$	$P_1(W)$	$P_1(C)$	$P_1(S)$
varmail	0.6% [0.2, 1.2]	0% [0, 0.2]	0.3% [0.1, 0.8]	99.1% [98.4, 99.5]
oltp	1.9% [1.2, 2.8]	0.1% [0, 0.5]	1.1% [0.6, 1.8]	96.9% [95.7, 97.8]
webserver	0.7% [0.3, 1.3]	1.4% [0.8, 2.2]	1.3% [0.7, 2.1]	97.0% [95.8, 97.9]
fileserver	7.1% [5.4, 9.0]	3.6% [2.5, 4.8]	1.6% [1.0, 2.5]	92.4% [90.2, 94.2]

Table 6: $P_{16}(X)$ **and** $P_1(X)$**.** *The upper table presents percentage values of the probabilities and 95% confidence intervals (in square brackets) of reading corrupt data (R), writing corrupt data (W), crash/hang and everything being fine (S), given that 16 bits are flipped, on a machine of 2GB memory. The lower table gives the derived percentage values given that 1 bit is corrupted. The working set size of each workload is less than 2GB; the average amount of page cache consumed by each workload after the bit flips are injected is 31MB (varmail), 129MB (oltp), 441MB (webserver) and 915MB (fileserver).*

Table 6 provides the probabilities and confidence intervals given that 16 bits are flipped and the derived values given that 1 bit is flipped. Note that for each workload, the sum of $P_k(R)$, $P_k(W)$, $P_k(C)$ and $P_k(S)$ is not necessary equal to 1, because there are cases where multiple failure scenarios occur in one trial.

From the lower table in Table 6, we see that a single bit flip in memory causes a small but non-negligible percentage of runs to experience an failure. For all workloads, the probability of reading corrupt data is greater than 0.6% and the probability of crashing or hanging is higher than 0.3%. The probability of writing corrupt data varies widely from 0 to 3.6%. Our results also show that in most cases, when the working set size is less than the memory size, the more page cache the workload consumes, the more likely that a failure would occur if one bit is flipped.

In summary, when a single bit flip occurs, the chances of failure scenarios happening can not be ignored. Therefore, efforts should be made to preserve data integrity in memory and prevent these failures from happening.

7 Beyond ZFS

In addition to ZFS, we have applied the same fault injection framework used in Section 5 to a simpler filesystem, ext2. Our initial results indicate that ext2 is also vulnerable to memory corruptions. For example, corrupt data can be returned to the user or written to disk. When certain fields of a VFS inode are corrupted, operations on that inode fail or the whole system crashes. If the inode is dirty, the corrupted fields of the VFS inode are propagated to the inode in the page cache and are then written to disk, making the corruptions permanent. Moreover, if the superblock in the page cache is corrupted and flushed

to disk, it might result in an unmountable filesystem.

In summary, so far we have studied two extremes: ZFS, a complex filesystem with many techniques to maintain on-disk data integrity, and ext2, a simpler filesystem with few mechanisms to provide extra reliability. Both are vulnerable to memory corruptions. It seems that regardless of the complexity of the file system and the amount of machinery used to protect against disk corruptions, memory corruptions are still a problem.

8 Related work

Software-implemented fault injection techniques have been widely used to analyze the robustness of systems [10, 17, 25, 31, 48, 55]. For example, FINE used fault injection to emulate hardware and software faults in the operating system [31]; Weining et al. [25] injected faults to instruction streams of Linux kernel function to characterize Linux kernel behavior.

More recent works [5, 8, 44] have applied type-aware fault injection to analyze failure behaviors of different file systems to disk corruptions. Our analysis of on-disk data integrity in ZFS is similar to these studies.

Further, fault injection has also been used to analyze effects of memory corruptions on systems. FIAT [10] used fault injection to study the effects of memory corruptions in a distributed environment. Krishnan et al. applied a memory corruption framework to analyze the effects of metadata corruption on NFS [33]. Our study on in-memory data integrity is related to these studies in their goal of finding effects of memory corruptions.

However, our work on ZFS is the first comprehensive reliability analysis of local file system that covers carefully controlled experiments to analyze both on-disk and in-memory data integrity. Specifically, for our study of memory corruptions, we separately analyze ZFS behavior for faults in page cache metadata and data and for metadata structures in the heap. To the best of our knowledge, this is the first such comprehensive study of end-to-end file system data integrity.

9 Summary and discussion

In this paper, we analyzed a state-of-the-art file system, ZFS, to study the implications of disk and memory corruptions to data integrity. We used carefully controlled fault injection experiments to simulate realistic disk and memory errors and presented our observations about ZFS behavior and its robustness.

While the reliability mechanisms in ZFS are able to provide reasonable robustness against disk corruptions, memory corruptions still remain a serious problem to data integrity. Our results for memory corruptions indicate cases where bad data is returned to the user, operations silently fail, and the whole system crashes. Our probability analysis shows that one single bit flip has small but non-negligible chances to cause failures such as reading/writing corrupt data and system crashing.

We argue that file systems should be designed with end-to-end data integrity as a goal. File systems should not only provide protection against disk corruptions, but also aim to protect data from memory corruptions. Although dealing with memory corruptions is hard, we conclude by discussing some techniques that file systems can use to increase protection against memory corruptions.

Block-level checksums in the page cache: File systems could protect the vulnerable data and metadata blocks in the page cache by using checksums. For example, ZFS could use the checksums inside block pointers in the page cache, update them on block updates, and verify the checksums on reads. However, this does incur an overhead in computation as well as some complexity in implementation; these are always the tradeoffs one has to make for reliability.

Metadata checksums in the heap: Even with block-level checksums in the page cache, there are still copies of metadata structures in the heap that are vulnerable to memory corruptions. To provide end-to-end data integrity, data-structure checksums may be useful in protecting in-heap metadata structures.

Programming for error detection: Many serious effects of memory corruptions can be mitigated by using simple programming practices. One technique is to use existing redundancy in data structures for simple consistency checks. For instance, the case described in Observation 8 (Section 5.3) could be detected by comparing the expected level calculated from the `dn_levels` field of `dnode_phys_t` with the actual level stored inside the first block pointer. Another simple technique is to include magic numbers in metadata structures for sanity checking. For example, some "crash" cases happened due to bad block pointers obtained during the block tree traversal (Observation 7 in Section 5.3). Using a magic number in block pointers could help detect such cases and prevent unexpected behavior.

Acknowledgment

We thank the anonymous reviewers and Craig Soules (our shepherd) for their tremendous feedback and comments, which have substantially improved the content and presentation of this paper. We thank Asim Kadav for his initial work on ZFS on-disk fault injection. 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 grants: CCF-0621487, CNS-0509474, CNS-0834392, CCF-0811697, CCF-0811697, CCF-0937959, as well as by generous donations from NetApp, Inc, Sun Microsystems, and Google.

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] CERT/CC Advisories. http://www.cert.org/advisories/.

[2] Kernel Bug Tracker. http://bugzilla.kernel.org/.

[3] US-CERT Vulnerabilities Notes Database. http://www.kb.cert.org/vuls/.

[4] D. Anderson, J. Dykes, and E. Riedel. More Than an Interface: SCSI vs. ATA. In *FAST*, 2003.

[5] L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Dependability Analysis of Virtual Memory Systems. In *DSN*, 2006.

[6] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *SIGMETRICS*, 2007.

[7] L. N. Bairavasundaram, G. R. Goodson, B. Schroeder, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *FAST*, 2008.

[8] L. N. Bairavasundaram, M. Rungta, N. Agrawal, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. M. Swift. Analyzing the Effects of Disk-Pointer Corruption. In *DSN*, 2008.

[9] W. Bartlett and L. Spainhower. Commercial Fault Tolerance: A Tale of Two Systems. *IEEE Trans. on Dependable and Secure Computing*, 1(1), 2004.

[10] J. Barton, E. Czeck, Z. Segall, and D. Siewiorek. Fault Injection Experiments Using FIAT. *IEEE Trans. on Comp.*, 39(4), 1990.

[11] R. Baumann. Soft errors in advanced computer systems. *IEEE Des. Test*, 22(3):258–266, 2005.

[12] E. D. Berger and B. G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *PLDI*, 2006.

[13] J. Bonwick. RAID-Z. http://blogs.sun.com/bonwick/entry/raid_z.

[14] J. Bonwick and B. Moore. ZFS: The Last Word in File Systems. http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf.

[15] F. Buchholz. The structure of the Reiser file system. http://homes.cerias.purdue.edu/∼florian/reiser/reiserfs.php.

[16] R. Card, T. Ts'o, and S. Tweedie. Design and Implementation of the Second Extended Filesystem. In Proceedings of the First Dutch International Symposium on Linux, 1994.

[17] J. Carreira, H. Madeira, and J. G. Silva. Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers. *IEEE Trans. on Software Engg.*, 1998.

[18] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault Containment for Shared-Memory Multiprocessors. In *SOSP*, 1995.

[19] C. L. Chen. Error-correcting codes for semiconductor memories. *SIGARCH Comput. Archit. News*, 12(3):245–247, 1984.

[20] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating System Errors. In *SOSP*, 2001.

[21] N. Dor, M. Rodeh, and M. Sagiv. CSSV: towards a realistic tool for statically detecting all buffer overflows in C. In *PLDI*, 2003.

[22] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *SOSP*, 2001.

[23] A. Eto, M. Hidaka, Y. Okuyama, K. Kimura, and M. Hosono. Impact of neutron flux on soft errors in mos memories. In *IEDM*, 1998.

[24] R. Green. EIDE Controller Flaws Version 24. http://mindprod.com/jgloss/eideflaw.html.

[25] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z. Yang. Characterization of Linux Kernel Behavior Under Errors. In *DSN*, 2003.

[26] H. S. Gunawi, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. SQCK: A Declarative File System Checker. In *OSDI*, 2008.

[27] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *PLDI*, 2002.

[28] J. Hamilton. Successfully Challenging the Server Tax. http://perspectives.mvdirona.com/2009/09/03/SuccessfullyChallengingTheServerTax.aspx.

[29] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *USENIX Winter*, 1992.

[30] D. T. J. A white paper on the benefits of chipkill- correct ecc for pc server main memory. *IBM Microelectronics Division*, 1997.

[31] W. Kao, R. K. Iyer, and D. Tang. FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior Under Faults. In *IEEE Trans. on Software Engg.*, 1993.

[32] A. Krioukov, L. N. Bairavasundaram, G. R. Goodson, K. Srinivasan, R. Thelen, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Parity Lost and Parity Regained. In *FAST*, 2008.

[33] S. Krishnan, G. Ravipati, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and B. P. Miller. The Effects of Metadata Corruption on NFS. In *StorageSS*, 2007.

[34] X. Li, K. Shen, M. C. Huang, and L. Chu. A memory soft error measurement on production systems. In *USENIX*, 2007.

[35] T. C. May and M. H. Woods. Alpha-particle-induced soft errors in dynamic memories. *IEEE Trans. on Electron Dev*, 26(1), 1979.

[36] N. Megiddo and D. Modha. Arc: A self-tuning, low overhead replacement cache. In *FAST*, 2003.

[37] R. C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO*, 1987.

[38] D. Milojicic, A. Messer, J. Shau, G. Fu, and A. Munoz. Increasing relevance of memory hardware errors: a case for recoverable programming models. In *ACM SIGOPS European Workshop*, 2000.

[39] B. Moore. Ditto Blocks - The Amazing Tape Repellent. http://blogs.sun.com/bill/entry/ditto_blocks_the_amazing_tape.

[40] E. Normand. Single event upset at ground level. *Nuclear Science, IEEE Transactions on*, 43(6):2742–2750, 1996.

[41] T. J. O'Gorman, J. M. Ross, A. H. Taber, J. F. Ziegler, H. P. Muhlfeld, C. J. Montrose, H. W. Curtis, and J. L. Walsh. Field testing for cosmic ray soft errors in semiconductor memories. *IBM J. Res. Dev.*, 40(1):41–50, 1996.

[42] Oracle Corporation. Btrfs: A Checksumming Copy on Write Filesystem. http://oss.oracle.com/projects/btrfs/.

[43] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *SIGMOD*, 1988.

[44] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON File Systems. In *SOSP*, 2005.

[45] F. Qin, S. Lu, and Y. Zhou. Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs. In *HPCA*, 2005.

[46] B. Schroeder, E. Pinheiro, and W.-D. Weber. DRAM errors in the wild: a large-scale field study. In *SIGMETRICS*, 2009.

[47] T. J. Schwarz, Q. Xin, E. L. Miller, D. D. Long, A. Hospodor, and S. Ng. Disk Scrubbing in Large Archival Storage Systems. In *MASCOTS*, 2004.

[48] D. Siewiorek, J. Hudak, B. Suh, and Z. Segal. Development of a Benchmark to Measure System Robustness. In *FTCS-23*, 1993.

[49] C. A. Stein, J. H. Howard, and M. I. Seltzer. Unifying File System Protection. In *USENIX*, 2001.

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

[51] Sun Microsystems. ZFS On-Disk Specification. http://www.opensolaris.org/os/community/zfs/docs/ondiskformat0822.pdf.

[52] R. Sundaram. The Private Lives of Disk Drives. http://partners.netapp.com/go/techontap/matl/sample/0206tot_resiliency.html.

[53] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. In *SOSP*, 2003.

[54] The Data Clinic. Hard Disk Failure. http://www.dataclinic.co.uk/hard-disk-failures.htm.

[55] T. K. Tsai and R. K. Iyer. Measuring Fault Tolerance with the FTAPE Fault Injection Tool. In *The 8th International Conference On Modeling Techniques and Tools for Computer Performance Evaluation*, 1995.

[56] S. C. Tweedie. Journaling the Linux ext2fs File System. In *The Fourth Annual Linux Expo*, Durham, North Carolina, 1998.

[57] J. Wehman and P. den Haan. The Enhanced IDE/Fast-ATA FAQ. http://thef-nym.sci.kun.nl/cgi-pieterh/atazip/atafq.html.

[58] G. Weinberg. The Solaris Dynamic File System. http://members.visi.net/∼thedave/sun/DynFS.pdf.

[59] A. Wenas. ZFS FAQ. http://blogs.sun.com/awenas/entry/zfs_faq.

[60] Y. Xie, A. Chou, and D. Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. In *FSE*, 2003.

[61] J. Yang, C. Sar, and D. Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *OSDI*, 2006.

[62] J. F. Ziegler and W. A. Lanford. Effect of cosmic rays on computer memories. *Science*, 206(4420):776–788, 1979.

Black-Box Problem Diagnosis in Parallel File Systems

Michael P. Kasick[1], Jiaqi Tan[2], Rajeev Gandhi[1], Priya Narasimhan[1]

[1] *Electrical & Computer Engineering Department*
Carnegie Mellon University
Pittsburgh, PA 15213–3890
{mkasick, rgandhi, priyan}@andrew.cmu.edu

[2] *DSO National Labs, Singapore*
Singapore 118230
tjiaqi@dso.org.sg

Abstract

We focus on automatically diagnosing different performance problems in parallel file systems by identifying, gathering and analyzing OS-level, black-box performance metrics on every node in the cluster. Our peer-comparison diagnosis approach compares the statistical attributes of these metrics across I/O servers, to identify the faulty node. We develop a root-cause analysis procedure that further analyzes the affected metrics to pinpoint the faulty resource (storage or network), and demonstrate that this approach works commonly across stripe-based parallel file systems. We demonstrate our approach for realistic storage and network problems injected into three different file-system benchmarks (dd, IOzone, and PostMark), in both PVFS and Lustre clusters.

1 Introduction

File systems can experience performance problems that can be hard to diagnose and isolate. Performance problems can arise from different system layers, such as bugs in the application, resource exhaustion, misconfigurations of protocols, or network congestion. For instance, Google reported the variety of performance problems that occurred in the first year of a cluster's operation [10]: 40–80 machines saw 50% packet-loss, thousands of hard drives failed, connectivity was randomly lost for 30 minutes, 1000 individual machines failed, etc. Often, the most interesting and trickiest problems to diagnose are not the outright crash (fail-stop) failures, but rather those that result in a "limping-but-alive" system (i.e., the system continues to operate, but with degraded performance). Our work targets the diagnosis of such performance problems in parallel file systems used for high-performance cluster computing (HPC).

Large scientific applications consist of compute-intense behavior intermixed with periods of intense parallel I/O, and therefore depend on file systems that can support high-bandwidth concurrent writes. Parallel Virtual File System (PVFS) [6] and Lustre [23] are open-source, parallel file systems that provide such applications with high-speed data access to files. PVFS and Lustre are designed as client-server architectures, with many clients communicating with multiple I/O servers and one or more metadata servers, as shown in Figure 1.

Problem diagnosis is even more important in HPC where the effects of performance problems are magnified due to long-running, large-scale computations. Current diagnosis of PVFS problems involve the manual analysis of client/server logs that record PVFS operations through code-level print statements. Such (white-box) problem diagnosis incurs significant runtime overheads, and requires code-level instrumentation and expert knowledge.

Alternatively, we could consider applying existing problem-diagnosis techniques. Some techniques specify a service-level objective (SLO) first and then flag runtime SLO violations—however, specifying SLOs might be hard for arbitrary, long-running HPC applications. Other diagnosis techniques first learn the normal (i.e., fault-free) behavior of the system and then employ statistical/machine-learning algorithms to detect runtime deviations from this learned normal profile—however, it might be difficult to collect fault-free training data for all of the possible workloads in an HPC system.

We opt for an approach that does not require the specification of an SLO or the need to collect training data for all workloads. We automatically diagnose performance problems in parallel file systems by analyzing the relevant black-box performance metrics on every node. Central to our approach is our hypothesis (borne out by observations of PVFS's and Lustre's behavior) that *fault-free I/O servers exhibit symmetric (similar) trends in their storage and network metrics, while a faulty server appears asymmetric (different) in comparison*. A similar hypothesis follows for the metadata servers. From these hypotheses, we develop a statistical peer-comparison approach that automatically diagnoses the faulty server and identifies the root cause, in a parallel file-system cluster.

The advantages of our approach are that it (i) exhibits *low overhead* as collection of OS-level performance metrics imposes low CPU, memory, and network demands; (ii) *minimizes training data* for typical HPC workloads by distinguishing between workload changes and performance problems with peer-comparison; and (iii) *avoids SLOs* by being agnostic to absolute metric values in identifying whether/where a performance problem exists.

We validate our approach by studying realistic storage and network problems injected into three file-system benchmarks (dd, IOzone, and PostMark) in two parallel file systems, PVFS and Lustre. Interestingly, but perhaps unsurprisingly, our peer-comparison approach identifies the faulty node even under workload changes (usually a source of false positives for most black-box problem-diagnosis techniques). We also discuss our experiences, particularly the utility of specific metrics for diagnosis.

2 Problem Statement

Our research is motivated by the following questions: (i) *can we diagnose the faulty server in the face of a performance problem in a parallel file system*, and (ii) *if so, can we determine which resource (storage or network) is causing the problem?*

Goals. Our approach should exhibit:

- *Application-transparency* so that PVFS/Lustre applications do not require any modification. The approach should be independent of PVFS/Lustre operation.
- *Minimal false alarms* of anomalies in the face of legitimate behavioral changes (e.g., workload changes due to increased request rate).
- *Minimal instrumentation overhead* so that instrumentation and analysis does not adversely impact PVFS/Lustre's operation.
- *Specific problem coverage* that is motivated by anecdotes of performance problems in a production parallel file-system deployment (see § 4).

Non-Goals. Our approach does not support:

- *Code-level debugging.* Our approach aims for coarse-grained problem diagnosis by identifying the culprit server, and where possible, the resource at fault. We currently do not aim for fine-grained diagnosis that would trace the problem to lines of PVFS/Lustre code.
- *Pathological workloads.* Our approach relies on I/O servers exhibiting similar request patterns. In parallel file systems, the request pattern for most workloads is similar across all servers—requests are either large enough to be striped across all servers or random enough to result in roughly uniform access. However, some workloads (e.g., overwriting the same portion of a file repeatedly, or only writing stripe-unit-sized records to every stripe-count offset) make requests distributed to only a subset, possibly one, of the servers.
- *Diagnosis of non-peers.* Our approach fundamentally cannot diagnose performance problems on non-peer nodes (e.g., Lustre's single metadata server).

Hypotheses. We hypothesize that, under a performance fault in a PVFS or Lustre cluster, OS-level performance metrics should exhibit observable anomalous behavior on the culprit servers. Additionally, with knowledge of PVFS/Lustre's overall operation, we hypothesize that the statistical trends of these performance data: (i) should be similar (albeit with inevitable minor differences) across fault-free I/O servers, even under workload changes, and (ii) will differ on the culprit I/O server, as compared to the fault-free I/O servers.

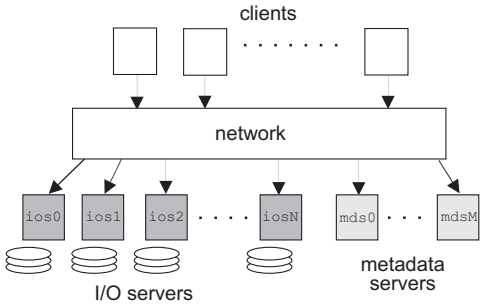


Figure 1: Architecture of parallel file systems, showing the I/O servers and the metadata servers.

Assumptions. We assume that a majority of the I/O servers exhibit fault-free behavior, that all peer server nodes have identical software configurations, and that the physical clocks on the various nodes are synchronized (e.g., via NTP) so that performance data can be temporally correlated across the system. We also assume that clients and servers are comprised of homogeneous hardware and execute homogeneous workloads. These assumptions are reasonable in HPC environments where homogeneity is both deliberate and critical to large scale operation. Homogeneity of hardware and client workloads is not strictly required for our diagnosis approach (§ 12 describes our experience with heterogeneous hardware). However we have not yet tested our approach with deliberately heterogeneous hardware or workloads.

3 Background: PVFS & Lustre

PVFS clusters consist of one or more metadata servers and multiple I/O servers that are accessed by one or more PVFS clients, as shown in Figure 1. The PVFS server consists of a single monolithic user-space daemon that may act in either or both metadata and I/O server roles.

PVFS clients consist of stand-alone applications that use the PVFS library (*libpvfs2*) or MPI applications that use the ROMIO MPI-IO library (that supports PVFS internally) to invoke file operations on one or more servers. PVFS can also plug in to the Linux Kernel's VFS interface via a kernel module that forwards the client's syscalls (requests) to a user-space PVFS client daemon that then invokes operations on the servers. This kernel client allows PVFS file systems to be mounted under Linux similar to other remote file systems like NFS.

With PVFS, file-objects are distributed across all I/O servers in a cluster. In particular, file data is striped

across each I/O server with a default stripe size of 64 kB. For each file-object, the first stripe segment is located on the I/O server to which the object handle is assigned. Subsequent segments are accessed in a round-robin manner on each of the remaining I/O servers. This characteristic has significant implications on PVFS's throughput in the event of a performance problem.

Lustre clusters consist of one active metadata server which serves one metadata target (storage space), one management server which may be colocated with the metadata server, and multiple object storage servers which serve one or more object storage targets each. The metadata and object storage servers are analogous to PVFS's metadata and I/O servers with the main distinction of only allowing for a single active metadata server per cluster. Unlike PVFS, the Lustre server is implemented entirely in kernel space as a loadable kernel module. The Lustre client is also implemented as a kernel space file-system module, and like PVFS, provides file system access via the Linux VFS interface. A userspace client library (*liblustre*) is also available.

Lustre allows for the configurable striping of file data across one or more object storage targets. By default, file data is stored on a single target. The `stripe_count` parameter may be set on a per-file, directory, or file-system basis to specify the number of object storage targets that file data is striped over. The `stripe_size` parameter specifies the stripe unit size and may be configured to multiples of 64 kB, with a default of 1 MB (the maximum payload size of a Lustre RPC).

4 Motivation: Real Problem Anecdotes

The faults we study here are motivated by the PVFS developers' anecdotal experience [5] of problems faced/reported in various production PVFS deployments, one of which is Argonne National Laboratory's 557 TFlop Blue Gene/P (BG/P) PVFS cluster. Accounts of experience with BG/P indicate that storage/network problems account for approximately 50%/50% of performance issues [5]. A single poorly performing server has been observed to impact the behavior of the overall system, instead of its behavior being averaged out by that of non-faulty nodes [5]. This makes it difficult to troubleshoot system-wide performance issues, and thus, fault localization (i.e., diagnosing the faulty server) is a critical first step in root-cause analysis.

Anomalous storage behavior can result from a number of causes. Aside from failing disks, RAID controllers may scan disks during idle times to proactively search for media defects [13], inadvertently creating disk contention that degrades the throughput of a disk array [25]. Our *disk-busy* injected problem (§ 5) seeks to emulate this manifestation. Another possible cause of a disk-busy problem is disk contention due to the accidental launch of a rogue processes. For example, if two remote file servers (e.g., PVFS and GPFS) are collocated, the startup of a second server (GPFS) might negatively impact the performance of the server already running (PVFS) [5].

Network problems primarily manifest in packet-loss errors, which is reported to be the "most frustrating" [sic] to diagnose [5]. Packet loss is often the result of faulty switch ports that enter a degraded state when packets can still be sent but occasionally fail CRC checks. The resulting poor performance spreads through the rest of the network, making problem diagnosis difficult [5]. Packet loss might also be the result of an overloaded switch that "just can't keep up" [sic]. In this case, network diagnostic tests of individual links might exhibit no errors, and problems manifest only while PVFS is running [5].

Errors do not necessarily manifest identically under all workloads. For example, SANs with large write caches can initially mask performance problems under write-intensive workloads and thus, the problems might take a while to manifest [5]. In contrast, performance problems in read-intensive workloads manifest rather quickly.

A consistent, but unfortunate, aspect of performance faults is that they result in a "limping-but-alive" mode, where system throughput is drastically reduced, but the system continues to run without errors being reported. Under such conditions, it is likely not possible to identify the faulty node by examining PVFS/application logs (neither of which will indicate any errors) [5].

Fail-stop performance problems usually result in an outright server crash, making it relatively easy to identify the faulty server. Our work targets the diagnosis of non-fail-stop performance problems that can degrade server performance without escalating into a server crash. There are basically three resources—CPU, storage, network—being contended for that are likely to cause throughput degradation. CPU is an unlikely bottleneck as parallel file systems are mostly I/O-intensive, and fair CPU scheduling policies should guarantee that enough time-slices are available. Thus, we focus on the remaining two resources, storage and network, that are likely to pose performance bottlenecks.

5 Problems Studied for Diagnosis

We separate problems involving storage and network resources into two classes. The first class is *hog* faults, where a rogue process on the monitored file servers induces an unusually high workload for the specific resource. The second class is *busy* or *loss* faults, where an unmonitored (i.e., outside the scope of the server OSes) third party creates a condition that causes a performance degradation for the specific resource. To explore all combinations of problem resource and class, we study the diagnosis of four problems—disk-hog, disk-busy, network-hog, packet-loss (network-busy).

Metric [s/n]*	Significance
tps [s]	Number of I/O (read and write) requests made to the disk per second.
rd_sec [s]	Number of sectors read from disk per second.
wr_sec [s]	Number of sectors written to disk per second.
avgrq-sz [s]	Average size (in sectors) of disk I/O requests.
avgqu-sz [s]	Average number of queued disk I/O requests; generally a low integer (0–2) when the disk is under-utilized; increases to ≈100 as disk utilization saturates.
await [s]	Average time (in milliseconds) that a request waits to complete; includes queuing delay and service time.
svctm [s]	Average service time (in milliseconds) of I/O requests; is the pure disk-servicing time; does not include any queuing delay.
%util [s]	Percentage of CPU time in which I/O requests are made to the disk.
rxpck [n]	Packets received per second.
txpck [n]	Packets transmitted per second.
rxbyt [n]	Bytes received per second.
txbyt [n]	Bytes transmitted per second.
cwnd [n]	Number of segments (per socket) allowed to be sent outstanding without acknowledgment.

*Denotes storage (s) or network (n) related metric.

Table 1: Black-box, OS-level performance metrics collected for analysis.

Disk-hogs can result from a runaway, but otherwise benign, process. They may occur due to unexpected cron jobs, e.g., an updatedb process generating a file/directory index for GNU locate, or a monthly software-RAID array verification check. Disk-busy faults can also occur in shared-storage systems due to a third-party/unmonitored node that runs a disk-hog process on the shared-storage device; we view this differently from a regular disk-hog because the increased load on the shared-storage device is not observable as a throughput increase at the monitored servers.

Network-hogs can result from a local traffic-emitter (e.g., a backup process), or the receipt of data during a denial-of-service attack. Network-hogs are observable as increased throughput (but not necessarily "goodput") at the monitored file servers. Packet-loss faults might be the result of network congestion, e.g., due to a network-hog on a nearby unmonitored node or due to packet corruption and losses from a failing NIC.

6 Instrumentation

For our problem diagnosis, we gather and analyze OS-level performance metrics, without requiring any modifications to the file system, the applications or the OS.

In Linux, OS-level performance metrics are made available as text files in the /proc pseudo file system. Table 1 describes the specific metrics that we collect. Most /proc data is collected via sysstat 7.0.0's sadc program [12]. sadc is used to periodically gather storage- and network-related metrics (as we are primarily concerned with performance problems due to storage and network resources, although other kinds of metrics are available) at a sampling interval of one second. For storage resources sysstat provides us with throughput (tps, rd_sec, wr_sec) and latency (await, svctm) metrics, and for network resources it provides us with throughput (rxpck, txpck, rxbyt, txbyt) metrics.

Unfortunately sysstat provides us only with throughput data for network resources. To obtain congestion data as well, we sample the contents of /proc/net/tcp, on both clients and servers, once every second. This gives us TCP congestion-control data [22] in the form of the sending congestion-window (cwnd) metric.

6.1 Parallel File-System Behavior

We highlight our (empirical) observations of PVFS's/Lustre's behavior that we believe is characteristic of stripe-based parallel file systems. Our preliminary studies of two other parallel file systems, GlusterFS [2] and Ceph [26], also reveal similar insights, indicating that our approach might apply to parallel file systems in general.

[Observation 1] *In a homogeneous (i.e., identical hardware) cluster, I/O servers track each other closely in throughput and latency, under fault-free conditions.*

For N I/O servers, I/O requests of size greater than $(N-1) \times stripe\_size$ results in I/O on each server for a single request. Multiple I/O requests on the same file, even for smaller request sizes, will quickly generate workloads[1] on all servers. Even I/O requests to files smaller than *stripe_size* will generate workloads on all I/O servers, as long as enough small files are read/written. We observed this for all three target benchmarks, dd, IOzone, and PostMark. For metadata-intensive workloads, we expect that metadata servers also track each other in proportional magnitudes of throughput and latency.

[Observation 2] *When a fault occurs on at least one of the I/O servers, the other (fault-free) I/O servers experience an identical drop in throughput.*

When a client syscall involves requests to multiple I/O servers, the client must wait for all of these servers to respond before proceeding to the next syscall.[2] Thus, the client-perceived cluster performance is constrained by the slowest server. We call this the *bottlenecking condition*. When a server experiences a performance fault, that server's per-request service-time increases. Because the

[1] Pathological workloads might not result in equitable workload distribution across I/O servers; one server would be disproportionately deluged with requests, while the other servers are idle, e.g., a workload that constantly rewrites the same *stripe_size* chunk of a file.

[2] Since Lustre performs client side caching and readahead, client I/O syscalls may return immediately even if the corresponding file server is faulty. Even so, a maximum of 32 MB may be cached (or 40 MB pre-read) before Lustre must wait for responses.

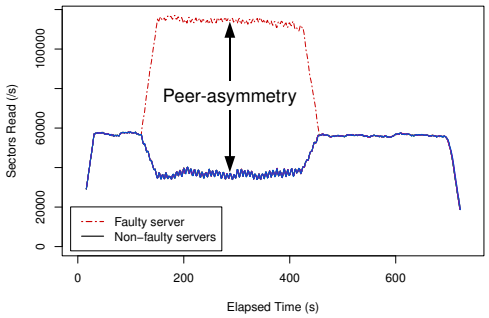


Figure 2: Peer-asymmetry of `rd_sec` for `iozoner` workload with *disk-hog* fault.

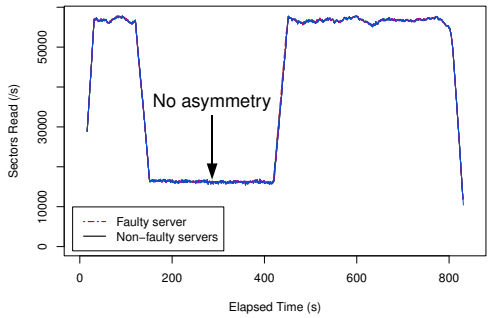


Figure 3: No asymmetry of `rd_sec` for `iozoner` workload with *disk-busy* fault.

client blocks on the syscall until it receives all server responses, the client's syscall-service time also increases. This leads to slower application progress and fewer requests per second from the client, resulting in a proportional decrease in throughput on all I/O servers.

[Observation 3] *When a performance fault occurs on at least one of the I/O servers, the other (fault-free) I/O servers are unaffected in their per-request service times.*

Because there is no server-server communication (i.e., no server inter-dependencies), a performance problem at one server will not adversely impact latency (per-request service-time) at the other servers. If these servers were previously highly loaded, latency might even improve (due to potentially decreased resource contention).

[Observation 4] *For disk/network-hog faults, storage/network-throughput increases at the faulty server and decreases at the non-faulty servers.*

A disk/network-hog fault at a server is due to a third-party that creates additional I/O traffic that is observed as increased storage/network-throughput. The additional I/O traffic creates resource contention that ultimately manifests as a decrease in file-server throughput on all servers (causing the bottlenecking condition of observation 2). Thus, disk- and network-hog faults can be localized to the faulty server by looking for *peer-divergence* (i.e. *asymmetry* across peers) in the storage- and network-throughput metrics, respectively, as seen in Figure 2.

[Observation 5] *For disk-busy (packet-loss) faults, storage- (network-) throughput decreases on all servers.*

For disk-busy (packet-loss) faults, there is no asymmetry in storage (network) throughputs across I/O servers (because there is no other process to create observable throughput, and the server daemon has the same throughput at all the nodes). Instead, there is a symmetric decrease in the storage-(network-) throughput metrics across all servers. Because asymmetry does not arise, such faults cannot be diagnosed, as seen in Figure 3.

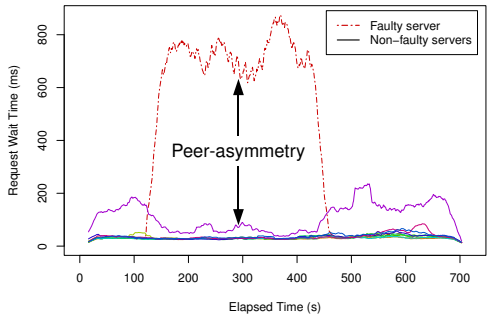


Figure 4: Peer-asymmetry of `await` for `ddr` workload with *disk-hog* fault.

[Observation 6] *For disk-busy and disk-hog faults, storage-latency increases on the faulty server and decreases at the non-faulty servers.*

For disk-busy and disk-hog faults, `await`, `avgqu-sz` and `%util` increase at the faulty server as the disk's responsiveness decreases and requests start to backlog. The increased `await` on the faulty server causes an increased server response-time, making the client wait longer before it can issue its next request. The additional delay that the client experiences reduces its I/O throughput, resulting in the fault-free servers having increased idle time. Thus, the `await` and `%util` metrics decrease asymmetrically on the fault-free I/O servers, enabling a peer-comparison diagnosis of the disk-hog and disk-busy faults, as seen in Figure 4.

[Observation 7] *For network-hog and packet-loss faults, the TCP congestion-control window decreases significantly and asymmetrically on the faulty server.*

The goal of TCP congestion control is to allow `cwnd` to be as large as possible, without experiencing packet-loss due to overfilling packet queues. When packet-loss occurs and is recovered within the retransmission timeout interval, the congestion window is halved. If recovery takes longer than retransmission timeout, `cwnd` is reduced to one segment. When nodes are transmitting data, their `cwnd` metrics either stabilize at high (≈100) val-

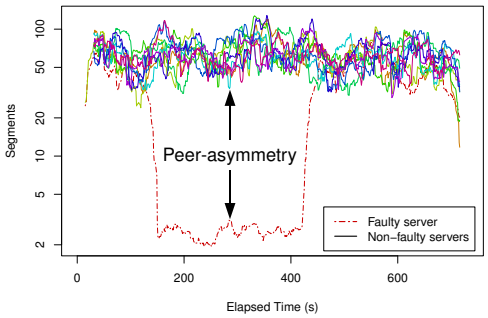


Figure 5: Peer-asymmetry of `cwnd` for `ddw` workload with *receive-pktloss* fault.

ues or oscillate (between ≈10–100) as congestion is observed on the network. However, during (some) network-hog and (all) packet-loss experiments, `cwnd`s of connections to the faulty server dropped by several orders of magnitude to single-digit values and held steady until the fault was removed, at which time the congestion window was allowed to open again. These asymmetric sustained drops in `cwnd` enable peer-comparison diagnosis for network faults, as seen in Figure 5.

7 Discussion on Metrics

Although faults present in multiple metrics, not all metrics are appropriate for diagnosis as they exhibit inconsistent behaviors. Here we describe problematic metrics.

Storage-throughput metrics. There is a notable relationship between the storage-throughput metrics: `tps` × `avgrq-sz` = `rd_sec` + `wr_sec`. While `rd_sec` and `wr_sec` accurately capture real storage activity and strongly correlate across I/O servers, `tps` and `avgrq-sz` do not correlate as strongly because a lower transfer rate may be compensated by issuing larger-sized requests. Thus, `tps` is not a reliable metric for diagnosis.

`svctm`. The impact of disk faults on `svctm` is inconsistent. The influences on storage service times are: time to locate the starting sector (seek time and rotational delay), media-transfer time, reread/rewrite time in the event of a read/write error, and delay time to due servicing of unobservable requests. During a disk fault, servicing of interleaved requests increases seek time. Thus, for an unchanged `avgrq-sz`, `svctm` will increase asymmetrically on the faulty server. Furthermore, during a disk-busy fault, servicing of unobservable requests further increases `svctm` due to request delays. However, during a disk-hog fault, the hog process might be issuing requests of smaller sizes than PVFS/Lustre. If so, then the associated decrease in media-transfer time might offset the increase in seek time resulting in a decreased or unchanged `svctm`. Thus, `svctm` is not guaranteed to exhibit asymmetries for disk-hogs, and therefore is unreliable.

Other metrics. While problems manifest on other metrics (e.g., CPU usage, context-switch rate), these secondary manifestations are due to the overall reduction in I/O throughput during the faulty period, and reveal nothing new. Thus, we do not analyze these metrics.

8 Experimental Set-Up

We perform our experiments on AMD Opteron 1220 machines, each with 4 GB RAM, two Seagate Barracuda 7200.10 320 GB disks (one dedicated for PVFS/Lustre storage), and a Broadcom NetXtreme BCM5721 Gigabit Ethernet controller. Each node runs Debian GNU/Linux 4.0 (etch) with Linux kernel 2.6.18. The machines run in stock configuration with background tasks turned off. We conduct experiments with x/y configurations, i.e., the PVFS x/y cluster comprises y combined I/O and metadata servers and x clients, while the equivalent Lustre x/y cluster comprises y object storage (I/O) servers with a single object storage target each, a single (dedicated) metadata server, and x clients. We conduct our experiments for 10/10 and 6/12 PVFS and Lustre clusters;[3] in the interests of space, we explain the 10/10 cluster experiments in detail, but our observations carry to both.

For these experiments PVFS 2.8.0 is used in the default server (`pvfs2-genconfig` generated) configuration with two modifications. First, we use the Direct I/O method (`TroveMethod directio`) to bypass the Linux buffer cache for PVFS I/O server storage. This is required for diagnosis as we otherwise observe disparate I/O server behavior during IOzone's rewrite phase. Although bypassing the buffer cache has no effect on diagnosis for non-rewrite (e.g., `ddw`) workloads, it does improve large write throughput by 10%.

Second, we increase to 4 MB (from 256 kB) the Flow buffer size (`FlowBufferSizeBytes`) to allow larger bulk data transfers and enable more efficient disk usage. This modification is standard practice in PVFS performance tuning, and is required to make our testbed performance representative of real deployments. It does not appear to affect diagnosis capability. In addition, we patch the PVFS kernel client to eliminate the 128 MB total size restriction on the `/dev/pvfs2-req` device request buffers and to `vmalloc` memory (instead of `kmalloc`) for the buffer page map (`bufmap_page_array`) to ensure that larger request buffers are actually allocatable. We then invoke the PVFS kernel client with 64 MB request buffers (`desc-size` parameter) in order to make the 4 MB data transfers to each of the I/O servers.

For Lustre experiments we use the etch backport of the Lustre 1.6.6 Debian packages in the default

[3] Due to a limited number of nodes we were unable to experiment with higher active client/server ratios. However, with the workloads and faults tested, an increased number of clients appears to degrade per-client throughput with no significant change in other behavior.

server configuration with a single modification to set the `lov.stripecount` parameter to −1 to stripe files across each object storage target (I/O server).

The nodes are rebooted immediately prior to the start of each experiment. Time synchronization is performed at boot-time using `ntpdate`. Once the servers are initialized and the client is mounted, monitoring agents start capturing metrics to a local (non-storage dedicated) disk. `sync` is then performed, followed by a 15-second sleep, and the experiment benchmark is run. The benchmark runs fault-free for 120 seconds prior to fault injection. The fault is then injected for 300 seconds and then deactivated. The experiment continues to the completion of the benchmark, which ideally runs for a total of 600 seconds in the fault-free case. This run time allows the benchmark to run for at least 180 seconds after a fault's deactivation to determine if there are any delayed effects. We run ten experiments for each workload & fault combination, using a different faulty server for each iteration.

8.1 Workloads

We use five experiment workloads derived from three experiment benchmarks: dd, IOzone, and PostMark. The same workload is invoked concurrently on all clients. The first two workloads, `ddw` and `ddr`, either write zeros (from `/dev/zero`) to a client-specific temporary file or read the contents of a previously written client-specific temporary file and write the output to `/dev/null`.

`dd` [24] performs a constant-rate, constant-workload large-file read/write from/to disk. It is the simplest large-file benchmark to run, and helps us to analyze and understand the system's behavior prior to running more complicated workloads. `dd` models the behavior of scientific-computing workloads with constant data-write rates.

Our next two workloads, `iozonew` and `iozoner`, consist of the same file-system benchmark, IOzone v3.283 [4]. We run `iozonew` in write/rewrite mode and `iozoner` in read/reread mode. IOzone's behavior is similar to `dd` in that it has two constant read/write phases. Thus, IOzone is a large-file I/O-heavy benchmark with few metadata operations. However, there is an `fsync` and a workload change half-way through.

Our fifth benchmark is PostMark v1.51 [15]. PostMark was chosen as a metadata-server heavy workload with small file writes (all writes < 64 kB thus, writes occur only on a single I/O server per file).

Configurations of Workloads. For the `ddw` workload, we use a 17 GB file with a record-size of 40 MB for PVFS, and a 30 GB file is used with a record-size 10 MB for Lustre. File sizes are chosen to result in a fault-free experiment runtime of approximately 600 seconds. The PVFS record-size was chosen to result in 4 MB bulk data transfers to each I/O server, which we empirically determined to be the knee of the performance vs. record-size curve. The Lustre record-size was chosen to result in 1 MB bulk data transfers to each I/O server—the maximum payload size of a Lustre RPC. Since Lustre both aggregates client writes and performs readahead, varying the record-size does not significantly alter Lustre read or write performance. For `ddr` we use a 27 GB file with a record-size of 40 MB for PVFS, and a 30 GB file with a record-size of 10 MB for Lustre (same as `ddw`).

For both the `iozonew` and `iozoner` workloads, we use an 8 GB file with a record-size of 16 MB (the largest that IOzone supports) for PVFS. For Lustre we use a 9 GB file with a record-size of 10 MB for `iozonew`, and a 16 GB file with the same record-size for `iozoner`. For `postmark` we use its default configuration with 16,000 transactions for PVFS and 53,000 transactions for Lustre to give a sufficiently long-running benchmark.

9 Fault Injection

In our fault-induced experiments, we inject a single fault at a time into one of the I/O servers to induce degraded performance for either network or storage resources. We inject the following faults:

- *disk-hog*: a `dd` process that reads 256 MB blocks (using direct I/O) from an unused storage disk partition.
- *disk-busy*: an `sgm_dd` process [11] that issues low-level SCSI I/O commands via the Linux SCSI Generic (`sg`) driver to read 1 MB blocks from the same unused storage disk partition.
- *network-hog*: a third-party node opens a TCP connection to a listening port on one of the PVFS I/O servers and sends zeros to it (*write-network-hog*), or an I/O server opens a connection and sends zeros to a third party node (*read-network-hog*).
- *pktloss*: a netfilter firewall rule that (probabilistically) drops packets received at one of the I/O servers with probability 5% (*receive-pktloss*), or a firewall rule on all clients that drops packets incoming from a single server with probability 5% (*send-pktloss*).

10 Diagnosis Algorithm

The first phase of the peer-comparison diagnostic algorithm identifies the faulty I/O server for the faults studied. The second phase performs root-cause analysis to identify the resource at fault.

10.1 Phase I: Finding the Faulty Server

We considered several statistical properties (e.g., the mean, the variance, etc. of a metric) as candidates for peer-comparison across servers, but ultimately chose the probability distribution function (PDF) of each metric because it captures many of the metric's statistical properties. Figure 6 shows the asymmetry in a metric's histograms/PDFs between the faulty and fault-free servers.

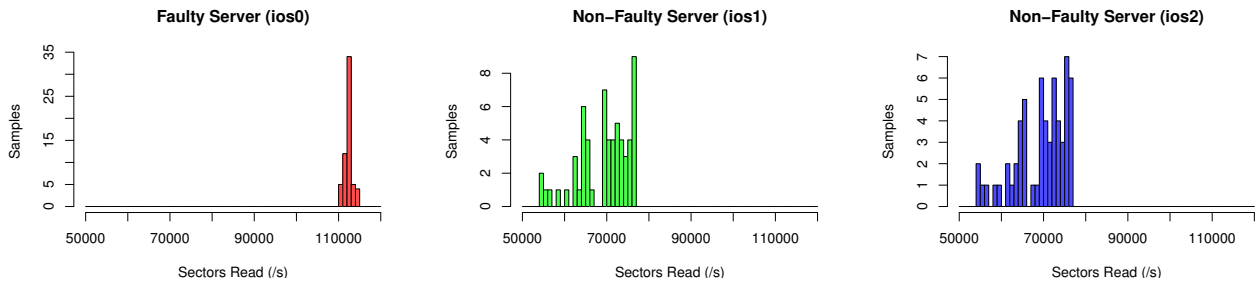


Figure 6: Histograms of `rd_sec` (`ddr` with *disk-hog* fault) for one faulty and two non-faulty servers.

Histogram-Based Approach. We determine the PDFs, using histograms as an approximation, of a specific black-box metric values over a window of time (of size *WinSize* seconds) at each I/O server. To compare the resulting PDFs across the different I/O servers, we use a standard measure, the Kullback-Leibler (KL) divergence [9], as the distance between two distribution functions, P and Q.[4] The KL divergence of a distribution function, Q, from the distribution function, P, is given by $D(P||Q) = \sum_i P(i) \log \frac{P(i)}{Qi}$. We use a symmetric version of the KL divergence, given by $D'(P||Q) = \frac{1}{2}[D(P||Q) + D(Q||P)]$ in our analysis.

We perform the following procedure for each of metric of interest. Using i to represent one of these metrics, we first perform a moving average on i. We then take PDFs of the smoothed i for two distinct I/O servers at a time and compute their pairwise KL divergences. A pairwise KL-divergence value for i is flagged as anomalous if it is greater than a certain predefined threshold. An I/O server is flagged as anomalous if its pairwise KL-divergence for i is anomalous with more than half of the other servers for at least k of the past $2k-1$ windows. The window is shifted in time by $WinShift$ (there is an overlap of $WinSize - WinShift$ samples between two consecutive windows), and the analysis is repeated. A server is indicted as faulty if it is anomalous in one or more metrics.

We use a 5-point moving average to ensure that metrics reflect average behavior of request processing. We also use a *WinSize* of 64, a $WinShift$ of 32, and a k of 3 in our analysis to incorporate a reasonable quantity of data samples per comparison while maintaining a reasonable diagnosis latency (approximately 90 seconds). We investigate the useful ranges of these values in § 11.2.

Time Series-Based Approach. We use the histogram-based approach for all metrics except `cwnd`. Unlike other metrics, `cwnd` tends to be noisy under normal conditions. This is expected as TCP congestion control prevents synchronized connections from fully utilizing link capacity. Thus `cwnd` analysis is different from other metrics as there is no closely-coupled peer behavior.

[4] Alternatively, earth mover's distance [20] or another distance measure may be used instead of KL.

Fortunately, there is a simple heuristic for detecting packet-loss using `cwnd`. TCP congestion control responds to packet-loss by halving `cwnd`, which results `cwnd` exponential decay after multiple loss events. When viewed on a logarithmic scale, sustained packet-loss results in a linear decrease for each packet lost.

To support analysis of `cwnd`, we first generate a time-series by performing a moving average on `cwnd` with a window size of 31 seconds. Based on empirical observation, this attenuates the effect of sporadic transmission timeout events while enabling reasonable diagnosis latencies (i.e., under one minute). Then, every second, a representative value (median) is computed of the log-`cwnd` values. A server is indicted if its log-`cwnd` is less than a predetermined fraction (threshold) of the median.

Threshold Selection. Both the histogram and time-series analysis algorithms require thresholds to differentiate between faulty and fault-free servers. We determine the thresholds through a fault-free training phase that captures a profile of relative server performance.

We do not need to train against all potential workloads, instead we train on workloads that are expected to stress the system to its limits of performance. Since server performance deviates the most when resources are saturated (and thus, are unable to "keep up" with other nodes), these thresholds represent the maximum expected performance deviations under normal operation. Less intense workloads, since they do not saturate server resources, are expected to exhibit better coupled peer behavior.

As the training phase requires training on the specific file system and hardware intended for problem diagnosis, we recommend training with HPC workloads normally used to stress-test systems for evaluation and purchase. Ideally these tests exhibit worst-case request rates, payload sizes, and access patterns expected during normal operation so as to saturate resources, and exhibit maximally-expected request queuing. In our experiments, we train with 10 iterations of the `ddr`, `ddw`, and `postmark` fault-free workloads. The same metrics are captured during training as when performing diagnosis.

To train the histogram algorithm, for each metric, we start with a minimum threshold value (currently 0.1) and

increase in increments (of 0.1) until the minimum threshold is determined that eliminates all anomalies on a particular server. This server-specific threshold is doubled to provide a cushion that masks minor manifestations occurring during the fault period. This is based on the premise that a fault's primary manifestation will cause a metric to be sufficiently asymmetric, roughly an order of magnitude, yielding a "safe window" of thresholds that can be used without altering the diagnosis.

Training the time-series algorithm is similar, except that the final threshold is not doubled as the `cwnd` metric is very sensitive, yielding a much smaller corresponding "safe window". Also, only two thresholds are determined for `cwnd`, one for all servers sending to clients, and one for clients sending to servers. As `cwnd` is generally not influenced by the performance of specific hardware, its behavior is consistent across nodes.

10.2 Phase II: Root-Cause Analysis

In addition to identifying the faulty server, we also infer the resource that is the root cause of the problem through an expert derived checklist. This checklist, based on our observations (§ 6.1) of PVFS's/Lustre's behavior, maps sets of peer-divergent metrics to the root cause. Where multiple metrics may be used, the specific metrics selected are chosen for consistency of behavior (see § 7). If we observe peer-divergence at any step of the checklist, we halt at that step and arrive at the root cause and faulty server. If peer-divergence is not observed at that step, we continue to the next step of decision-making.

Do we observe peer-divergence in ...

1. Storage throughput? (`rd_sec` or `wr_sec`) — Yes: disk-hog fault; No: next question
2. Storage latency? (`await`) — Yes: disk-busy fault; No: ...
3. Network throughput?* (`rxbyt` or `txbyt`) — Yes: network-hog fault; No: ...
4. Network congestion? (`cwnd`) — Yes: packet-loss fault; No: no fault discovered

*Must diverge in both `rxbyt` & `txbyt`, or in absence of peer-divergence in `cwnd` (see § 12).

11 Results

PVFS Results. Tables 2 and 3 shows the accuracy (true- and false-positive rates) of our diagnosis algorithm in indicting faulty nodes (ITP/IFP) and diagnosing root causes (DTP/DFP)[5] for the PVFS 10/10 & 6/12 clusters.

[5]ITP is the percentage of experiments where all faulty servers are correctly indicted as faulty, IFP is the percentage where at least one non-faulty server is misindicted as faulty. DTP is the percentage of experiments where all faults are successfully diagnosed to their root causes, DFP is the percentage where at least one fault is misdiagnosed to a wrong root cause (including misindictments).

Fault	ITP	IFP	DTP	DFP
None (control)	0.0%	0.0%	0.0%	0.0%
disk-hog	100.0%	0.0%	100.0%	0.0%
disk-busy	90.0%	2.0%	90.0%	2.0%
write-network-hog	92.0%	0.0%	84.0%	8.0%
read-network-hog	100.0%	0.0%	100.0%	0.0%
receive-pktloss	42.0%	0.0%	42.0%	0.0%
send-pktloss	40.0%	0.0%	40.0%	0.0%
Aggregate	77.3%	0.3%	76.0%	1.4%

Table 2: Results of PVFS diagnosis for the 10/10 cluster.

Fault	ITP	IFP	DTP	DFP
None (control)	0.0%	2.0%	0.0%	2.0%
disk-hog	100.0%	0.0%	100.0%	0.0%
disk-busy	100.0%	0.0%	100.0%	0.0%
write-network-hog	42.0%	2.0%	0.0%	44.0%
read-network-hog	0.0%	2.0%	0.0%	2.0%
receive-pktloss	54.0%	6.0%	54.0%	6.0%
send-pktloss	40.0%	2.0%	40.0%	2.0%
Aggregate	56.0%	2.0%	49.0%	8.0%

Table 3: Results of PVFS diagnosis for the 6/12 cluster.

It is notable that not all faults manifest equally on all workloads. *disk-hog*, *disk-busy*, and *read-network-hog* all exhibit a significant (> 10%) runtime increase for all workloads. In contrast, the *receive-pktloss* and *send-pktloss* only have significant impact on runtime for write-heavy and read-heavy workloads respectively. Correspondingly, faults with greater runtime impact are often the most reliably diagnosed. Since packet-loss faults have negligible impact on `ddr` & `ddw` ACK flows and `postmark` (where lost packets are recovered quickly), it is reasonable to expect to not be able to diagnose them.

When removing the workloads for which packet-loss cannot be observed (and thus, not diagnosed), the aggregate diagnosis rates improve to 96.3% ITP and 94.6% DTP in the 10/10 cluster, and to 67.2% ITP and 58.8% DTP in the 6/12 cluster.

Lustre Results. Tables 4 and 5 shows the accuracy of our diagnosis algorithm for the Lustre 10/10 & 6/12 clusters. When removing workloads for which packet-loss cannot be observed, the aggregate diagnosis rates improve to 92.5% ITP and 86.3% DTP in the 10/10 cluster, and to 90.0% ITP and 82.1% DTP in the 6/12 case.

Both 10/10 clusters exhibit comparable accuracy rates. In contrast, the PVFS 6/12 cluster exhibits masked network-hogs faults (fewer true-positives) due to low network throughput thresholds from training with unbalanced metadata request workloads (see § 12). The Lustre 6/12 cluster exhibits more misdiagnoses (higher false-positives) due to minor, secondary manifestations in storage throughput. This suggests that our analysis algorithm may be refined with a ranking mechanism that allows diagnosis to tolerate secondary manifestations (see § 14).

Fault	ITP	IFP	DTP	DFP
None (control)	0.0%	0.0%	0.0%	0.0%
disk-hog	82.0%	0.0%	82.0%	0.0%
disk-busy	88.0%	2.0%	68.0%	22.0%
write-network-hog	98.0%	2.0%	96.0%	4.0%
read-network-hog	98.0%	2.0%	94.0%	6.0%
receive-pktloss	38.0%	4.0%	36.0%	6.0%
send-pktloss	40.0%	0.0%	38.0%	2.0%
Aggregate	74.0%	1.4%	69.0%	5.7%

Table 4: Results of Lustre diagnosis for the 10/10 cluster.

Fault	ITP	IFP	DTP	DFP
None (control)	0.0%	6.0%	0.0%	6.0%
disk-hog	100.0%	0.0%	100.0%	0.0%
disk-busy	76.0%	8.0%	38.0%	46.0%
write-network-hog	86.0%	14.0%	86.0%	14.0%
read-network-hog	92.0%	8.0%	92.0%	8.0%
receive-pktloss	40.0%	2.0%	40.0%	2.0%
send-pktloss	38.0%	8.0%	38.0%	8.0%
aggregate	72.0%	6.6%	65.7%	12.0%

Table 5: Results of Lustre diagnosis for the 6/12 cluster.

11.1 Diagnosis Overheads & Scalability

Instrumentation Overhead. Table 6 reports runtime overheads for instrumentation of both PVFS and Lustre for our five workloads. Overheads are calculated as the increase in mean workload runtime (for 10 iterations) with respect to their uninstrumented counterparts. Negative overheads are result of sampling error, which is high due runtime variance across experiments. The PVFS workload with the least runtime variance (`iozoner`) exhibits, with 99% confidence, a runtime overhead < 1%. As the server load of this workload is comparable to the others, we conclude that OS-level instrumentation has negligible impact on throughput and performance.

Data Volume. The performance metrics collected by `sadc` have an uncompressed data volume of 3.8 kB/s on each server node, independent of workload or number of clients. The congestion-control metrics sampled from `/proc/net/tcp` have a data volume of 150 B/s per socket on each client & server node. While the volume of congestion-control data linearly increases with number of clients, it is not necessary to collect per-socket data for all clients. At minimum, congestion-control data needs to be collected for only a single active client per time window. Collecting congestion-control data from additional clients merely ensures that server packet-loss effects are observed by a representative number of clients.

Algorithm Scalability. Our analysis code requires, every second, 3.44 ms per server and 182 µs per server pair of CPU time on a 2.4 GHz dedicated core to diagnose a fault if any exists. Therefore, realtime diagnosis of up to 88 servers may be supported on a single 2.4 GHz core.

Although the pairwise analysis algorithm is $O(n^2)$, we recognize that it is not necessary to compare a given server against all others in every analysis window. To support very large clusters (thousands of servers), we recommend partitioning n servers into $n-k$ analysis domains of k (e.g., 10) servers each, and only performing pairwise comparisons within these partitions. To avoid undetected anomalies that might develop in static partitions, we recommend rotating partition membership in each analysis window. Although we have not yet tested this technique, it does allow for $O(n)$ scalability.

Overhead for Workload	File System	
	PVFS	Lustre
`ddr`	0.90% ± 0.62%	1.81% ± 1.71%
`ddw`	0.00% ± 1.03%	–0.22% ± 1.18%
`iozoner`	–0.07% ± 0.37%	0.70% ± 0.98%
`iozonew`	–0.77% ± 1.62%	0.53% ± 2.71%
`postmark`	–0.58% ± 1.49%	0.20% ± 1.28%

Table 6: Instrumentation overhead: Increase in runtime w.r.t. non-instrumented workload ± standard error.

11.2 Sensitivity

Histogram moving-average span. Due to large record sizes, some workload & fault combinations (e.g., `ddr` & disk-busy) yield request processing times up to 4 s. As client requests often synchronize (see § 12), metrics may reflect distinct request processing stages instead of aggregate behavior. For example, during a disk fault, the faulty server performs long, low-throughput storage operations while fault-free servers perform short, high-throughput operations. At 1 s resolution, these behaviors reflect asymmetrically in many metrics. While this feature results in high (79%) ITP rates, its presence in nearly all metrics results in high (10%) DFP rates as well. Furthermore, since the influence of this feature is dependent on workload and number of clients, it is not reliable, and therefore, it is important to perform metric smoothing.

However, "too much" smoothing eliminates medium-term variances, decreasing TP and increasing FP rates. With 9-point smoothing, DFP (11%) exceeds unsmoothed while DTP reduces by 11% to 58.3%. Therefore we chose 5-point smoothing to minimize IFP (2.4%) and DFP (6.7%) with a modest decrease in DTP (64.9%).

Anomalous window filtering. In histogram-based analysis, servers are flagged anomalous only if they demonstrate anomalies in k of the past $2k-1$ windows. This filtering reduces false-positives in the event of sporadic anomalous windows when no underlying fault is present. k in the range 3–7 exhibits a consistent 6% increase in ITP/DTP and a 1% decrease in IFP/DFP over the non-filtered case. For $k \geq 8$, the TP/FP rates decrease/increase again. We expect k's useful-range upper-bound to be a function of the time that faults manifest.

`cwnd` moving-average span. For `cwnd` analysis a moving average is performed on the time series to atten-

uate the effect of sporadic transmission timeouts. This enforces the condition that timeout events sustain for a reasonable time period, similar to anomalous window filtering. Spans in the range 5–31, with 31 the largest tested, exhibit a consistent 8% increase in ITP/DTP and a 1% decrease in IFP/DFP over the non-smoothed case.

WinSize & WinShift. Seven *WinSize*s of 32–128 with 16 sample steps, and seven *WinShift*s of 16–64 with 8 sample steps were tested to determine diagnosis influence. All *WinSize*s $\geq$ 48 and *WinShift*s $\geq$ 32 were comparable in performance (62–66% DTP, 6–9% DFP). Thus for sufficiently large values, diagnosis is not sensitive.

Histogram threshold scale factor. Histogram thresholds are scaled by a factor (currently 2x) to provide a cushion against secondary, minor fault manifestations (see § 10.1). At 1x, FP rates increase to 19%/23% IFP/DFP. 1.5x reduces this to 3%/8% IFP/DFP. On the range 2–4x ITP/DTP decreases from 70%/65% to 54%/48% as various metrics are masked, while IFP/DFP hold at 2%/7% as no additional misdiagnoses occur.

12 Experiences & Lessons

We describe some of our experiences, highlighting counterintuitive or unobvious issues that arose.

Heterogeneous Hardware. Clusters with heterogeneous hardware will exhibit performance characteristics that might violate our assumptions. Unfortunately, even supposedly homogeneous hardware (same make, model, etc.) can exhibit slightly different performance behaviors that impede diagnosis. These differences mostly manifest when the devices are stressed to performance limits (e.g., saturated disk or network).

Our approach can compensate for some deviations in hardware performance as long as our algorithm is trained for stressful workloads where these deviations manifest. The tradeoff, however, is that performance problems of lower severity (whose impact is less than normal deviations) may be masked. Additionally, there may be factors that are non-linear in influence. For example, buffer-cache thresholds are often set as a function of the amount of free memory in a system. Nodes with different memory configurations will have different caching semantics, with associated non-linear performance changes that cannot be easily accounted for during training.

Multiple Clients. Single- vs. multi-client workloads exhibit performance differences. In PVFS clusters with caching enabled, the buffer cache aggregates contiguous small writes for single-client workloads, considerably improving throughput. The buffer cache is not as effective with small writes in multi-client workloads, with the penalty due to interfering seeks reducing throughput and pushing disks to saturation.

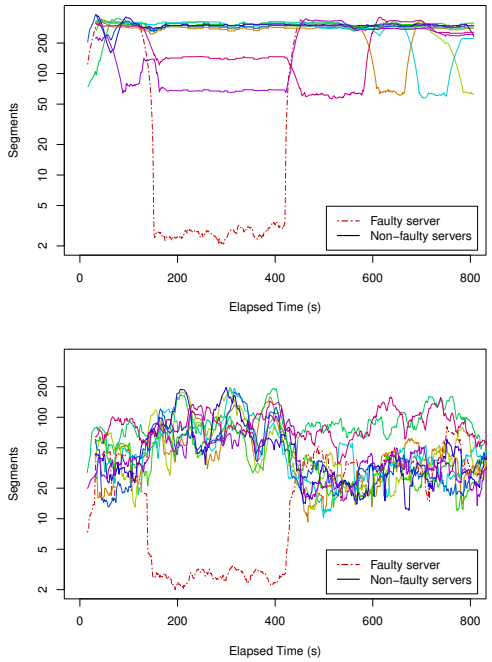


Figure 7: Single (top) and multiple (bottom) client `cwnd`s for `ddw` workloads with *receive-pktloss* faults.

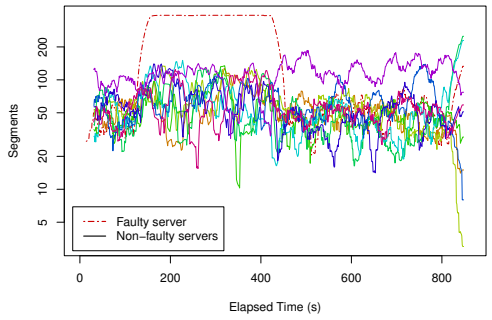


Figure 8: Disk-busy fault influence on faulty server's `cwnd` for `ddr` workload.

This also impacts network congestion (see Figure 7). Single-client write workloads create single-source bulk data transfers, with relatively little network congestion. This creates steady client `cwnd`s that deviate sharply during a fault. Multi-client write workloads create multi-source bulk data transfers, leading to interference, congestion and chaotic, widely varying `cwnd`s. While a faulty server's `cwnd`s are still distinguishable, this highlights the need to train on stressful workloads.

Cross-Resource Fault Influences. Faults can exhibit cross-metric influence on a single resource, e.g., a disk-hog creates increased throughput on the faulty disk, saturating that disk, increasing request queuing and latency.

Faults affecting one resource can manifest unintuitively in another resource's metrics. Consider a disk-busy fault's influence on the faulty server's `cwnd` for a

large read workload (see Figure 8). cwnd is updated only when a server is both sending and experiencing congestion; thus, cwnd does not capture the degree of network congestion when a server is *not* sending data. Under a disk-busy fault, (i) a single client would send requests to each server, (ii) the fault-free servers would respond quickly and then idle, and (iii) the faulty server would respond after a delayed disk-read request.

PVFS' lack of client read-ahead blocks clients on the faulty server's responses, effectively synchronizing clients. Bulk data transfers occur in phases (ii) and (iii). During phase (ii), all fault-free servers transmit, creating network congestion and chaotic cwnd values, whereas during phase (iii), only the faulty server transmits, experiencing almost no congestion and maintaining a stable, high cwnd value. Thus, the faulty server's cwnd is asymmetric w.r.t. the other servers, mistakenly indicating a network-related fault instead of a disk-busy fault.

We can address this by assigning greater weight to storage-metric anomalies over network-metric anomalies in our root-cause analysis (§ 10.2). With Lustre's client read-ahead, read calls are not as synchronized across clients, and this influence does not manifest as severely.

Metadata Request Heterogeneity. Our peer-similarity hypothesis does not apply to PVFS metadata servers. Specifically, since each PVFS directory entry is stored in a single server, server requests are unbalanced during path lookups, e.g., the server containing the directory "/" is involved in nearly all lookups, becoming a bottleneck.

We address this heterogeneity by training on the postmark metadata-heavy workload. Unbalanced metadata requests create a spread in network-throughput metrics for each server, contributing to a larger training threshold. If the request imbalance is significant, the resulting large threshold for network-throughput metrics will mask nearly all network-hog faults.

Buried ACKs. Read/write-network-hogs induce deviations in both receive and send network-throughput due to the network-hog's payload and associated acknowledgments. Since network-hog ACK packets are smaller than data packets, they can easily be "buried" in the network-throughput due to large-I/O traffic. Thus, network-hogs can appear to influence only one of rxbyt or txbyt, for read or write workloads, respectively.

rxpck and txpck metrics are immune to this effect, and can be used as alternatives for rxbyt and txbyt for network-hog diagnosis. Unfortunately, the non-homogeneous nature of metadata operations (in particular, postmark) result in rxpck/txpck fault manifestations being masked in most circumstances.

Delayed ACKs. In contradiction to Observation 5, a receive-(send-) packet-loss fault during a large-write (large-read) workload can cause a steady receive (send) network throughput on the faulty node and asymmetric decreases on non-faulty nodes. Since the receive (send) throughput is almost entirely comprised of ACKs, this phenomenon is the result of delayed ACK behavior.

Delayed ACKs reduce ACK traffic by acknowledging every other packet when packets are received in order, effectively halving the amount of ACK traffic that would otherwise be needed to acknowledge packets 1:1. During packet-loss, each out-of-order packet is acknowledged 1:1 resulting in an effective doubling of receive (send) throughput on the faulty server as compared to non-faulty nodes. Since the packet-loss fault itself results in, approximately, a halving of throughput, the overall behavior is a steady or slight increase in receive (sent) throughput on the faulty node during the fault period.

Network Metric Diagnosis Ambiguity. A single network metric is insufficient for diagnosis of network faults because of three properties of network throughput and congestion. First, *write-network-hog*s during write workloads create enough congestion to deviate the client cwnd; thus, cwnd is not an exclusive indicator of a packet-loss fault. Second, delayed ACKs contribute to packet-loss faults manifesting as network-throughput deviations, on rxbyt or txbyt; thus, the absence of a throughput deviation in the presence of a cwnd does not sufficiently diagnose all packet-loss faults. Third, buried ACKs contribute to network-hog faults manifesting in only one of rxbyt and txbyt, but not both; thus, the presence of both rxbyt and txbyt deviations does not sufficiently indicate all network-hog faults.

Thus, we disambiguate network faults in the third root-cause analysis step as follows. If both rxbyt and txbyt are asymmetric across servers, regardless of cwnd, a network-hog fault exists. If either rxbyt or txbyt is asymmetric, in the absence of cwnd, a network-hog fault exists. If cwnd is asymmetric regardless of either rxbyt or txbyt (but not both, due to the first rule above), then a packet-loss fault exists.

13 Related Work

Peer-comparison Approaches. Our previous work [14] utilizes a syscall-based approach to diagnosing performance problems in addition to propagated errors and crash/hang problems in PVFS. Currently, the performance metric approach described here is capable of more accurate diagnosis of performance problems with superior root-cause determination as compared to the syscall-based approach, although the syscall approach is capable of diagnosing non-performance problems in PVFS that would otherwise escape diagnosis here. The syscall-based approach also has a significantly higher worst-observed runtime overhead (≈65%) and per-server data volumes on the order of 1 MB/s, raising performance and

scalability concerns in larger deployments.

Ganesha [18], seeks to diagnose performance-related problems in Hadoop by classifying slave nodes, via clustering of performance metrics, into behavioral profiles which are then peer-compared to indict nodes behaving anomalously. While the node indictment methods are similar, our work peer-compares a limited set of performance metrics directly (without clustering), which enables us to attribute the affected metrics to a root-cause. In contrast, Ganesha is limited to identifying faulty nodes only, it does not perform root-cause analysis.

The closest non-authored work is Mirgorodskiy et al. [17], which localizes code-level problems by tracing function calls and peer comparing their execution times across nodes to identify anomalous nodes in an HPC cluster. As a debugging tool, it is designed to locate the specific functions where problems manifest in cluster software. The performance problems studied in our work tend to escape diagnosis with their technique as the problems manifest in increased time spent in the file servers' descriptor poll loop that is symmetric across faulty and fault-free nodes. Thus, our work aims to target the resource responsible for performance problems.

Metric Selection. Cohen et al. [8] uses a statistical approach to metric selection for problem diagnosis in large systems with many available metrics by identifying those with a high efficacy at diagnosing SLO violations. They achieve this by a summary and index of system history as expressed by the available metrics and by marking signatures of past histories as being indicative of a particular problem, which enables them to diagnose future occurrences. Our metric selection is expert-based, since in the absence of SLOs, we must determine which metrics reliably peer-compare to determine if a problem exists. We also select metrics based on semantic relevance, so that we can attribute asymmetries to behavioral indications of particular problems that hold across different clusters.

Message-based Problem Diagnosis. Many previous works have focused on path-based [1, 19, 3] and component-based [7, 16] approaches to problem diagnosis in Internet Services. Aguilera et al. [1] treats components in a distributed system as black-boxes, inferring paths by tracing RPC messages and detecting faults by identifying request flow paths with abnormally long latencies. Pip [19] traces causal request flows with tagged messages, which are checked against programmer-specified expectations. Pip identifies requests and specific lines of code as faulty when they violate these expectations. Magpie [3] uses expert knowledge of event orderings to trace causal request flows in a distributed system. Magpie then attributes system resource utilizations (e.g. memory, CPU) to individual requests and clusters them by their resource usage profiles to detect faulty requests. Pinpoint [7, 16] tags request flows through J2EE web-service systems, and, once a request is known to have failed, it identifies the responsible request processing components.

Each of the path- and component-based approaches rely on tracing of intercomponent messages (e.g., RPCs) as the primary means of instrumentation. This requires either modification of the messaging libraries (which, for parallel file systems is usually contained in server application code) or, at minimum, the ability to sniff messages and extract features from them. Unfortunately, the message interfaces used by parallel file systems are often proprietary and insufficiently documented, making such instrumentation difficult. Hence, our initial attempts to diagnose problems in parallel file systems specifically avoid message-level tracing by identifying anomalies through peer-comparison of global performance metrics.

While performance metrics are lightweight and easy to obtain, we believe that traces of component-level messages (i.e., client requests & responses) would serve as a rich source of behavioral information, and would prove beneficial in diagnosing problems with subtler manifestations. With the recent standardization of Parallel NFS [21] as a common interface for parallel storage, future adoption of this protocol would encourage investigation of message-based techniques in our problem diagnosis.

14 Future Work

We intend to improve our diagnosis algorithm by incorporating a ranking mechanism to account for secondary fault manifestations. Although our threshold selection is good at determining whether a fault exists at all in the cluster, if a fault presents in two metrics with significantly different degrees of manifestation, then our algorithm should place precedence on the metric with the greater manifestation instead of indicting one arbitrarily.

In addition, we intend to validate our diagnosis approach on a large HPC cluster with a significantly increased client/server ratio and real scientific workloads to demonstrate our diagnosis capability at scale. We intend to expand our problem coverage to include more complex sources of performance faults. Finally, we intend to expand our instrumentation to include additional black-box metrics as well as client request tracing.

15 Conclusion

We presented a black-box problem-diagnosis approach for performance faults in PVFS and Lustre. We have also revealed our (empirically-based) insights about PVFS's and Lustre's behavior with regard to performance faults, and have used these observations to motivate our analysis approach. Our fault-localization and root-cause analysis identifies both the faulty server and the resource at fault, for storage- and network-related problems.

Acknowledgements

We thank our shepherd, Gary Grider, for his comments that helped us to improve this paper. We also thank Rob Ross, Sam Lang, Phil Carns and Kevin Harms of Argonne National Laboratory for their insightful discussions on PVFS, instrumentation and troubleshooting, and anecdotes of problems in production deployments. This research was sponsored in part by NSF grants #CCF–0621508 and by ARO agreement DAAD19–02–1–0389.

References

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 74–89, Bolton Landing, NY, Oct. 2003.

[2] A. Babu. GlusterFS, Mar. 2009. `http://www.gluster.org/`.

[3] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 259–272, San Francisco, CA, Dec. 2004.

[4] D. Capps. IOzone filesystem benchmark, Oct. 2006. `http://www.iozone.org/`.

[5] P. H. Carns, S. J. Lang, K. N. Harms, and R. Ross. Private communication, Dec. 2008.

[6] P. H. Carns, W. B. Ligon, and R. B. R. andRajeev Thakur. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, Atlanta, GA, Oct. 2000.

[7] M. Y. Chen, E. Kıcıman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, Bethesda, MD, June 2002.

[8] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, Brighton, UK, Oct. 2005.

[9] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley-Interscience, New York, NY, Aug. 1991.

[10] J. Dean. Underneath the covers at Google: Current systems and future directions, May 2008.

[11] D. Gilbert. The Linux sg3_utils package, June 2008. `http://sg.danny.cz/sg/sg3_utils.html`.

[12] S. Godard. SYSSTAT utilities home page, Nov. 2008. `http://pagesperso-orange.fr/sebastien.godard/`.

[13] D. Habas and J. Sieber. Background Patrol Read for Dell PowerEdge RAID Controllers. *Dell Power Solutions*, Feb. 2006.

[14] M. P. Kasick, K. A. Bare, E. E. Marinelli III, J. Tan, R. Gandhi, and P. Narasimhan. System-call based problem diagnosis for PVFS. In *Proceedings of the 5th Workshop on Hot Topics in System Dependability*, Lisbon, Portugal, June 2009.

[15] J. Katcher. PostMark: A new file system benchmark. Technical Report TR3022, Network Appliance, Inc., Oct. 1997.

[16] E. Kıcıman and A. Fox. Detecting application-level failures in component-based Internet services. *IEEE Transactions on Neural Networks*, 16(5):1027–1041, Sept. 2005.

[17] A. V. Mirgorodskiy, N. Maruyama, and B. P. Miller. Problem diagnosis in large-scale computing environments. In *Proceedings of the ACM/IEEE conference on Supercomputing*, Tampa, FL, Nov. 2006.

[18] X. Pan, J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan. Ganesha: Black-box diagnosis of mapreduce systems. In *Proceedings of the 2nd Workshop on Hot Topics in Measurement & Modeling of Computer Systems*, Seattle, WA, June 2009.

[19] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, , and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *Proceedings of the 3rd Conference on Networked Systems Design and Implementation*, San Jose, CA, May 2006.

[20] Y. Rubner, C. Tomasi, and L. J. Guibas. A metric for distributions with applications to image databases. In *Proceedings of the 6th International Conference on Computer Vision*, pages 59–66, Bombay, India, Jan. 1998.

[21] S. Shepler, M. Eisler, and D. Noveck. NFS version 4 minor version 1. Internet-Draft, Dec. 2008.

[22] W. R. Stevens. TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms. RFC 2001 (Proposed Standard), Jan. 1997.

[23] Sun Microsystems, Inc. Lustre file system: High-performance storage architecture and scalable cluster file system. White paper, Oct. 2008.

[24] The IEEE and The Open Group. dd, 2004. `http://www.opengroup.org/onlinepubs/009695399/utilities/dd.html`.

[25] J. Vasileff. latest PERC firmware == slow, July 2005. `http://lists.us.dell.com/pipermail/linux-poweredge/2005-July/021908.html`.

[26] S. A. Weil, S. A. Brandt, E. L. Miller, and D. D. E. Long. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 307–320, Seattle, WA, Nov. 2006.

A Clean-Slate Look at Disk Scrubbing

Alina Oprea
RSA Laboratories
Cambridge, MA
aoprea@rsa.com

Ari Juels
RSA Laboratories
Cambridge, MA
ajuels@rsa.com

Abstract

A number of techniques have been proposed to reduce the risk of data loss in hard-drives, from redundant disks (e.g., RAID systems) to error coding within individual drives. Disk scrubbing is a background process that reads disks during idle periods to detect irremediable read errors in infrequently accessed sectors. Timely detection of such latent sector errors (LSEs) is important to reduce data loss.

In this paper, we take a clean-slate look at disk scrubbing. We present the first formal definition in the literature of a scrubbing algorithm, and translate recent empirical results on LSE distributions into new scrubbing principles. We introduce a new simulation model for LSE incidence in disks that allows us to optimize our proposed scrubbing techniques and demonstrate the significant benefits of intelligent scrubbing to drive reliability. We show how optimal scrubbing strategies depend on disk characteristics (e.g., the BER rate), as well as disk workloads.

1 Introduction

With the unremitting growth of digital information in the world, there is an ever increasing reliance on hard drives for critical data storage. Hard drives serve not only as primary storage devices, but due to their growing capacity and dropping prices, they are now an attractive building block for a range of storage systems, including large-scale secondary systems (e.g., archival or backup systems). In these environments, their reliability becomes significant and needs to be quantified, as some of these systems demand strict and high availability guarantees.

A significant body of research focuses on designing reliable storage systems by adding redundant disks. RAID systems enhance reliability by storing parity blocks in redundant arrays. Most systems today employ RAID-5 or RAID-6 mechanisms that are resilient to one or two simultaneous disk failures, respectively. Data loss in RAID is amplified by *latent sector errors* (LSEs), sector errors in drives that are not detected when they occur, but only when the disk area is accessed in the normal course of use. In RAID-5, a disk failure coupled with only one latent error on another disk induces data loss.

To increase the reliability of both single drives and RAID systems, researchers have studied techniques such as *intra-disk redundancy* [5] or *disk scrubbing* [15]. Intra-disk redundancy applies an erasure code over a subset (segment) of consecutive sectors in the drive and stores the parity blocks in the same disk. It protects against a small number of LSEs in each segment, depending on the parameters of the erasure code.

Disk scrubbing is a background process that reads disk sectors during idle periods, with the goal of detecting latent sector errors in infrequently accessed blocks. Most existing systems perform *sequential* disk scrubbing, meaning that they access disk sectors by increasing logical block address, and use a scrubbing rate that is constant or dependent on the amount of disk idle time. Mi et al. [9], for instance, suggest that disk scrubbing should be scheduled whenever the disk is idle in order to maximize scrubbing rates. A notable exception is the work of Schwarz et al. [15], which considers alternative scrubbing strategies with varying rates; the goal is to minimize disk power-on time in large archival systems whose disks are generally powered off.

In this paper, we define the first formal model for scrubbing strategies, along with a performance metric for the single-drive setting. Through a simulation model, we empirically search the space of scrubbing strategies and find optimal points in this space. We translate new results in the literature on the distribution of LSEs in hard drives

[2] into new scrubbing principles. The main message of the paper is that by exploiting a richer design space for scrubbing strategies, we can design better algorithms that significantly improve current technologies. We have to note, though, that our results are highly sensitive to some disk parameters that are not always made public by disk manufacturers. We hope that this paper will open up a new line of research that will further refine our results as more accurate disk failure data becomes available to the community.

In more detail, our main technical contributions are:

Formal model for scrubbing strategies We give the first formal model for scrubbing strategies that considers a number of disk parameters (e.g., disk age, disk model, disk failure rates), as well as history of disk usage. We view a scrubbing strategy as a function which, given information about a drive, outputs the set of sectors to be scrubbed in the next time interval.

The metrics most commonly used for hard drive reliability are MTTF (Mean Time To Failure) for single drives, and MTTDL (Mean Time To Data Loss) for a RAID system. For single drive reliability, MTTF measures the disk lifetime before total failure, and does not give a measure of its resilience to LSEs. MTTDL is a systemic measure, and not applicable to the study of errors in a single drive. Thus we define a new metric for hard drives called MLET ("Mean Latent Error Time"). MLET captures the percentage of time in which the disk is susceptible to data loss due to an LSE (and can serve as a basis for determining MTTDL). We define an optimal scrubbing strategy for a drive to be one that minimizes our new MLET metric.

Latent-sector error model Based on the results presented by Bairavasundaram et al. [2], and known results about usage-related LSEs [6], we propose a simple model for LSE development. Our model considers both age-related and usage-related LSEs, and captures their *spatial and temporal locality*. Since we do not have complete information about LSE distribution from the academic literature, we derive additional assumptions to generate a complete LSE model. We show that our model accurately reflects the field data presented by Bairavasundaram et al. We believe that our model is of general interest in the study of LSEs, as it provides a simplified and efficient tool for experimentation.

Find optimal strategy through simulation Guided by new empirical results on LSE distributions in the literature, we identify new scrubbing principles for single disks, summarized in Table 1. These principles suggest several new dimensions in the formulation of scrubbing strategies (e.g., variable scrubbing rates) and lead us to a newly enriched design space. Using a simulation based on our proposed LSE model, we search this design space for MLET-optimal scrubbing strategies. We find an optimal scrubbing strategy which, compared with straightforward sequential scrubbing, improves on the MLET metric by an order of magnitude.

Organization We review related work in Section 2. We create a model for the distribution of LSEs using the study of Bairavasundaram et al. [2] and additional assumptions, and validate this model against the study's empirical data in Section 3. We define scrubbing strategies formally, introduce our new design dimensions, and formulate our search space for scrubbing strategies in Section 4. We describe our simulation model and present our results on simulation-optimized scrubbing strategies in Section 5. We conclude in Section 6.

2 Related Work

Several recently published papers have shifted the storage community's perspective on disk failures in the real world. Schroeder and Gibson [14] show that annual disk failure rates are higher than those published by manufacturers, and determine that disks do not exhibit exponential times between failures (as commonly believed). Instead, time between failures is modeled more accurately by a Weibull distribution. Pinheiro et al. [11] offer statistics on disk survival rates conditioned on various SMART parameters. The first study on latent sector errors (LSEs) for field data is that of Bairavasundaram et al. [2]. They show that LSE rates increase linearly with disk age, and that LSEs are highly correlated, exhibiting both spatial and temporal locality.

Disk scrubbing is a well known technique used extensively to detect latent sector errors early. Most existing systems use a sequential scrubbing strategy in which sectors are read from disk in increasing order of their logical address. In the academic literature, more sophisticated scrubbing strategies have been proposed by Schwartz et al. [15] in the context of large archival storage systems. In such systems, one goal is to keep the disk powered down as much as possible, and minimize the number of power ups. Their opportunistic strategy piggybacks on normal read accesses—scrubbing when a disk is powered up for another operation. They also propose a simple, three-state Markov model that captures disk degradation due to scrubbing. Within this analytic model, they

Facts about LSE distribution	Corresponding proposed scrubbing principles
1. LSE rate is low in the first 60 days of operation	1. Keep scrubbing rate low during the first 60 days of operation
2. After 60 days, LSE rate is higher, but fairly constant before the first LSE develops	2. After 60 days, increase scrubbing rate and keep it constant before detecting a first LSE
3. LSEs exhibit temporal locality	3. Increase scrubbing rate after LSE detection
4. LSEs exhibit spatial locality	4. Staggered scrubbing (defined in Section 4.2) is superior to sequential or randomized scrubbing
5. LSEs develop as a function of disk usage	5. Scrubbing is not free: limit scrubbing rate to avoid collateral LSEs

Table 1: Translation of results on LSEs in the literature into scrubbing principles

calculate the optimal scrubbing rate.

To the best of our knowledge, our work provides the first general formalization of scrubbing strategies for hard drives and optimizes such strategies over a large search space. In contrast to Schwartz et al., we are interested in enterprise disks that are powered up most of the time, and we do not consider the power-up effect on reliability. Interestingly, we observe the adverse effect of aggressive scrubbing, much like Schwartz et al. While in [15], aggressive scrubbing detrimentally increases the number of disk power ups, in our system aggressive scrubbing triggers LSEs by increasing disk usage. Through our newly defined MLET metric, we are able to capture the effect of usage errors for drive reliability. We thus dispute the common belief that scrubbing is most effective at maximum capacity.

A number of research papers examine the effect of scrubbing and LSEs on RAID reliability. In his Ph.D. thesis [8], Kari developed the first Markov model for RAID reliability that considers LSEs (in addition to total disk failures). He obtained theoretical equations for MTTDL (the RAID reliability metric defined by Patterson et al. [10]), assuming that the distribution of LSEs is exponential. More recently, Elerath and Pecht [6] propose a 5-state simulation model for RAID-5, in which both the disk failure and LSE distributions are modeled by a Weibull probability density function.

Baker et al. [3] provide a reliability model for two-way mirroring in the context of long-term archival storage. In their Markov model, they consider exponentially distributed LSEs and their spatial and temporal correlation, which they model via an increased rate in their exponential distribution. They also show that scrubbing at a constant rate (every two weeks) reduces MTTDL.

Beyond scrubbing, there exist other single-disk techniques to protect against LSEs. Intra-disk redundancy schemes (IDR) [5] encode additional redundancy *within the disk itself* in the form of erasure codes. Dholakia et al. [5] propose encoding consecutive disk sectors under a custom-crafted XOR erasure code. Iliadis et al. [7] compare disk scrubbing and IDR with respect to RAID reliability. Mi et al. [9] consider the problem of scheduling background activities, including scrubbing and IDR, to increase the MTTDL metric for RAID. They show that combining scrubbing and IDR greatly improves RAID reliability.

3 Modeling the Distribution of Latent Sector Errors

We model the distribution of latent sector errors (LSEs) using the data presented in the recent NetApp study of Bairavasundaram et al. [2]. The NetApp study is the only published academic paper that gives a substantial characterization of LSE development. That said, the paper does not contain or reference detailed data: The LSE-development data sets on which the paper is based are proprietary, and have not been publicly released. Given these facts, our only choice to derive a meaningful LSE model was to reverse engineer some of the graphs presented in the NetApp paper. We make additional assumptions about LSE development as needed to generate a complete LSE model. We validate our LSE model against the graphs provided by the NetApp paper, but, of course, thorough validation of the model requires access to real data.

3.1 Results from NetApp study

The NetApp study [2] presents results on the LSE distribution of 1.53 million disks from various models and manufacturers over a 24-month period. The disks are divided into two classes: nearline and enterprise. In our work here, though, we restrict our study to enterprise disks. The main findings of the NetApp study on enterprise disks are summarized below:

1. LSEs develop at a fairly constant rate in the first two years of a drive's age. An exception are the first two months; these exhibit a slightly lower LSE rate. The fraction of disks developing at least one LSE is highly variable for different disk models, ranging at the end of the 24-month study from 1% to 4%.

2. LSEs exhibit *spatial locality* at the logical address level, as shown by two graphs in the paper. Figure 5 from the NetApp study shows the probability of another error within a given radius of an existing LSE. For most disk models, the probability of another latent error within 10MB of an existing error is 0.5. Figure 6 from the NetApp study shows the average number of errors within a given radius of an existing error. While both graphs provide some information about how LSEs are clustered together, the NetApp study does not provide full details about the exact probability distribution function of LSE locations in disks.

3. LSEs exhibit *temporal locality*. More than 80% of errors arrive at an interval of less than an hour from previous errors. Figure 7 in [2] shows that the inter-arrival time distribution has very long tails.

4. As shown in Figure 8 of [2], most additional errors occur in the first month after the first LSE, and the probability of developing these errors decays exponentially over time. For instance, the probability of a disk developing 1, 10, and 50 additional errors in the first month is 0.6, 0.25 and 0.1, respectively.

3.2 Latent sector error model

The NetApp study shows how latent errors develop in disks as a function of disk age. We call such errors *age errors*. Additionally, latent errors develop due to disk usage or disk wear-out. A hard-drive metric that captures usage is the *byte-error rate* (BER). While there is no consensus in the literature on the interpretation of this metric [4], we assume that both reads and writes contribute to development of usage errors, albeit with different weights. In our disk model, we vary the BER metric between 10^{-15} and 10^{-13} (to capture disks with various characteristics), and we define a read/write weight for each disk, denoted RW_Weight (to characterize the relative contribution of read and write operations to disk wear-out). We refer to the errors that develop due to disk wear-out as *usage errors*.

There is no explicit information in the academic literature about the exact distribution of usage-related LSEs. Since it is very likely that during the 24-month NetApp study at least several usage-related LSEs developed, we make the assumption that usage-related LSEs follow a spatial and temporal distribution similar to age errors.

The NetApp study shows that LSEs are clustered both spatially and temporally. We further categorize age and usage LSEs into two types of errors. The first type is that of *triggering errors*. We define a triggering error to be either the first age-related error in a drive, or the first usage-related error that develops after a specified amount of data has been accessed (counting from the time the previous usage-related error developed). A triggering error induces a cluster of additional errors, called *triggered errors*. These errors develop in a short interval of time after the corresponding triggering error, and are clustered spatially on disk closely to the triggering error.

Before giving full details on our LSE model, let us start with some intuition on modeling the spatial and temporal distribution of LSEs.

Modeling spatial distribution on disk As the NetApp study observes, most LSEs are clustered at radii of around 10-100MB. We define the *centroid* of a cluster to be the median error in the cluster with respect to block logical addresses. In our simulation model in Section 5, we need to generate errors in increasing order of occurrence time. For convenience in that model, we assume that the triggering error (i.e., the first error in a cluster) is also the cluster centroid. Since the NetApp study does not provide the exact location on disk of error clusters (but only error relative distance), we assume that the centroid location is uniformly distributed across all disk sectors. We model the triggered errors as being clustered around the centroid with radii determined from the distribution given in Figure 5 of [2]. In Section 3.3, we regenerate the graphs presenting spatial locality of LSEs in the NetApp study using our LSE model, in order to validate our simplifying assumptions.

Modeling temporal distribution We model the time at which a triggering error develops after the data in the NetApp study. Figure 1 in [2] gives the probability that a disk develops an age error in its first 24 months in the field; the results are presented at the granularity of six months. Combined with the results from Figure 10 in [2], we infer that the disk error rate is lower in the first 60 days of disk operation, and fairly constant after that. In our simulation model, we work at the temporal granularity of one hour. Without finer granularity on how triggering age errors develop temporally, we assume that the time a disk develops its first LSE error is uniformly distributed within the month in which the triggering error arises.

The time a usage error develops is determined by the disk BER metric, which we vary between 10^{-15} and 10^{-13}. We assume that usage error development follows a normal distribution with mean $1/\text{BER}$. A usage error is triggered once the number of bytes accessed (due to both normal disk workloads and the scrubbing process) weighted by RW_Weight, exceeds on average $1/\text{BER}$.

Once the occurrence time of the centroid is determined, we generate the number of additional errors in the disk based on the graph from Figure 8 in [2]. Figure 8 gives the probability of a disk developing up to 50 errors after a first LSE. The NetApp study does not provide a maximum limit on the number of LSEs in a disk, but it states that about 80% of disks develop less than 50 errors. We set the maximum number of LSEs in the disk to 100. The inter-arrival time for each triggered error is modeled with the distribution from Figure 7 in [2].

To generate the distributions from Figures 1, 5 and 7 in the NetApp paper we used piecewise uniform distributions with points given by those graphs. For Figure 8, we used curve fitting in Mathematica.

We summarize the assumptions made in generating our LSE model in Table 2.

1. Age errors form a single cluster on disk.
2. Usage error clusters develop due to both reads and writes, albeit with different weights.
3. Usage error clusters follow spatial and temporal correlations similar to those exhibited by age errors.
4. Development of a new triggering usage error follows a normal distribution with mean $1/\text{BER}$ and small deviation.
5. The triggering error of an error cluster is the cluster centroid.
6. Triggered errors developing closely in time are clustered around the centroid.
7. Cluster centroids are uniformly distributed on disk.
8. The time a triggering error develops in a month is uniformly distributed within the month.

Table 2: Assumptions for generating LSE model.

Formally, we define an LSE model as a probability distribution function $\mathcal{P}_{\mathsf{LSE}}$. First, let us define a bit vector E_t over all sectors in the disk, such that $E_t(s) = 1$ if sector s has developed a latent sector error at time t and $E_t(s) = 0$, otherwise. Taking as input time t, sector s, the cumulative write and read usage up to time t in bytes, denoted W_t and R_t, respectively, and the history of latent error development $E_1, \ldots, E_{t-1}$, $\mathcal{P}_{\mathsf{LSE}}(t, s, W_t, R_t, E_1, \ldots, E_{t-1})$ is the probability that sector s develops a latent sector error at time t. Let us denote the space of all LSE models as $\mathcal{L}$.

We give now full details on our LSE model.

1. **Modeling triggering age LSE.** Using Figures 1 and 10 from [2], we determine the probability that a disk develops an age error in each month of its first 24 months in the field. If a disk develops a triggering error in month $0 \leq m \leq 23$, then the exact occurrence time in hours is uniformly generated in the month, according to the distribution $U(720 * m, 720 * (m + 1) - 1)$. (Here $U(a, b)$ is the uniform distribution on $[a, b]$.)

2. **Modeling triggering usage LSE.** We fix the BER metric for a disk to a value in the set $\{10^{-15}, 10^{-14.5}, 10^{-14}, 10^{-13.5}, 10^{-13}\}$. Once the BER metric is fixed (e.g., 10^{-14}), a usage error is developed when Bytes_Written + Bytes_Read/RW_Weight $>= 1/\text{BER}$. If we use a fixed value for BER in the above equation, we get a fixed trigger time of usage errors, which results in a very restrictive model. We instead randomize usage error development: we assume that $1/\text{BER}$ is just the mean of the number of bytes accessed after the disk develops an usage error, and we assume that usage error development follows a normal distribution with mean $1/\text{BER}$ and small variance σ (e.g., 20% of the mean). We first generate a Gaussian random variable $X \sim N(1/\text{BER}, \sigma)$, and then trigger a usage error once Bytes_Written + Bytes_Read/RW_Weight $>= X$. For the read/write weight RW_Weight we use values between 1 and 9.

3. **Location of triggering error.** Assuming that a disk develops a triggering error (either age or usage) at time t_c (expressed in hours), we determine its exact location l_c on disk as a uniformly distributed random variable over all disk sectors.

4. **Number of triggered errors.** We determine the number of triggered LSE from Figure 8 in [2]. Using curve fitting in Mathematica, we determine that the probability that a disk develops x triggered errors is given (approximately) by the function $f(x) = 1.04x^{-0.185} - 0.42$.

5. **Location of triggered LSEs.** We assume that the triggered LSEs are clustered around the triggering error, with a relative distance following the piecewise uniform distribution from Figure 5 in the NetApp study.

6. **Time of triggered LSEs.** The inter-arrival time for each LSE from the previous one in the cluster is modeled with the piecewise uniform distribution from Figure 7 in the NetApp study.

We list the range of parameters used in our LSE model in Table 3.

Parameter	Range/value	Justification
Max number of errors	100	[2]
BER	$[10^{-15}, 10^{-13}]$	[6]
RW_Weight	[1,9]	Heuristic assumption
Deviation σ of usage error development	20% of mean	Heuristic assumption

Table 3: Parameter ranges in LSE model.

3.3 Model validation

We perform several experiments to validate our LSE model. We generate age-related LSEs for 100,000 disks

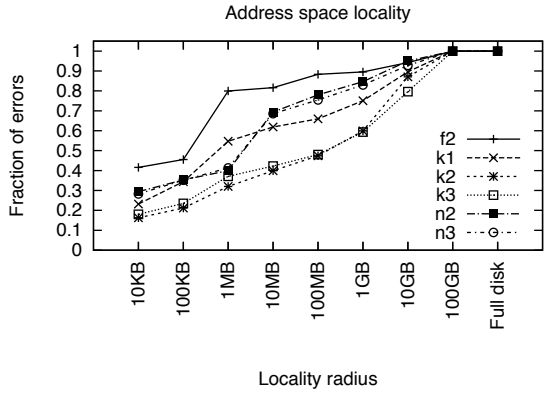


Figure 1: Fraction of errors within a given radius of an existing LSE in our simulation model.

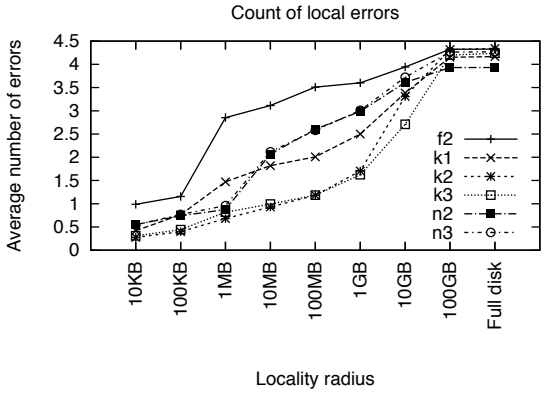


Figure 2: Average number of errors within a given radius of an existing LSE in our simulation model.

using our model and based on Figures 1, 5, 7, 8 and 10 of the NetApp study. While Figures 8 and 10 represent distributions for all disk models, Figures 1, 5 and 7 give different distributions depending on the disk model. There are six different enterprise models common to these three figures (denoted f-2, k-1, k-2, k-3, n-2 and n-3). These disk models are anonymized in the NetApp paper and we do not have information about exact disk characteristics. According to the NetApp study, drives labeled with the same letter have the same (anonymized) manufacturer, and a higher number denotes higher drive capacity (e.g., k_1, k_2 and k_3 have the same manufacturer and increasing capacities).

As monthly error rates and inter-arrival time for age errors in our simulation are generated exactly as in the NetApp study, we focus on validating our spatial LSE model. Our main goal is to validate assumptions we make due to incomplete data in the distribution of LSE location on disk, as explained above. For that, we regenerate graphs from Figures 5 and 6 in the NetApp study after the location of age errors is generated with our simulation model. Note that the results from Figure 6 are not used in our simulation model at all.

As in Figures 5 and 6 in [2], Figure 1 shows the probability of a new error arising within a given radius of an existing error, and Figure 2 shows the average number of errors within a given radius of an LSE, for the six disk models described above.

We observe that our simulation model closely reflects the results from the NetApp study. For disk models that exhibit high locality (e.g., f-2), the results of the simulation are within 1% of the study results. For models with a lower degree of locality, our simulation model slightly over-estimates the two metrics, but our simulation results differ by 6% on average from the study results.

Due to its simplicity and accuracy, we believe our LSE model is of general and practical value in the study of LSEs.

4 Scrubbing Strategies

In this section, we give the first formalization of scrubbing strategies in the literature that takes into account information about the disk model and its history. Most systems today use a simple constant-rate sequential scrubbing strategy. To capture the spatial and temporal locality of LSE development, we expand the space of scrubbing strategies across several dimensions. First, we propose a *staggered* strategy that traverses disk regions more rapidly than sequential reading. Thanks to the spatial locality of LSEs, it discovers LSEs faster than sequential scrubbing. We evaluate the performance impact of staggering, and determine parameters for which its overhead—resulting from frequent disk-head movement—is minimal (2%) compared with sequential scrubbing. Second, we consider scrubbing strategies that adaptively change their scrubbing rate according to drive age and the history of LSE development. Based on these new ideas, we propose an expanded design space of scrubbing strategies.

4.1 Formal Definition

Our formalization of scrubbing strategies accounts for disk model and age, as well as historical factors, including disk usage, the number of developed latent errors, and the scrubbing history.

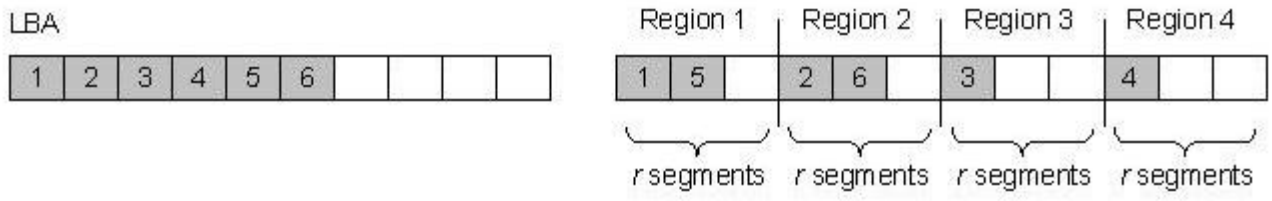


Figure 3: Representation of sequential (left) and staggered (right) scrubbing strategies.

Formally, we define a scrubbing strategy as a function of the disk age t, cumulative disk write and read usage, latent error distribution, disk failure distribution, latent error development history and scrubbing history. This function outputs the number and addresses of sectors to be scrubbed in the current time interval t.

Definition 1. A scrubbing strategy for a disk with n sectors is a function S. For inputs disk age t, cumulative disk write W_t and read usage R_t, latent error distribution $\mathcal{P}_{\mathsf{LSE}} \in \mathcal{L}$, disk failure distribution $\mathcal{P}_{\mathsf{DF}}$ in space $\mathcal{F}$, latent error development history $L_t^h = \{E_1, \ldots, E_{t-1}\}$ (as defined in Section 3.2), and scrubbing history $S_t^h = \{v_i, [1,n]^{v_i}\}_{i=1,\ldots t-1}$ (including the number and addresses of sectors scrubbed at all previous time intervals), it outputs the number of sectors selected for scrubbing v_t, and their logical block addresses $(\mathsf{LBA}_1, \ldots, \mathsf{LBA}_{v_t})$.

For example, assuming that LBAs are between 0 and $n-1$, the sequential strategy with constant-rate r can be formally defined as $S(t, W_t, R_t, \mathcal{P}_{\mathsf{LSE}}, \mathcal{P}_{\mathsf{DF}}, L_t^h, S_t^h) = \{r, (rt+1 \bmod n, \ldots, r(t+1) \bmod n)\}$. Note that the constant-rate sequential strategy only depends on disk age, but it does not take into account other disk characteristics or history of error development.

We leave the definition of the disk failure distribution as general as possible. It can depend on disk age, disk usage and failure history, similar to the definition of LSE distribution. We omit the disk failure history from the scrubbing strategy definition since once a disk fails, it is replaced with a new one and our model is restarted.

4.2 Staggered scrubbing

Our staggered scrubbing regime—again, aimed at exploiting the spatial locality of LSEs—is as follows. The disk is partitioned into m regions, each consisting of r segments. Staggered scrubbing reads the first segment of each disk region in turn, ordered by LBA. Then it reads the second segment in each disk region, and so forth, up to the r^{th} segment, as depicted in Figure 3. (Once a full scrubbing pass is complete, it is initiated again with the first segment.)

Intuitively, staggering is effective because LSEs tend to arise in clusters: if a given region develops LSEs, there is a good chance that many of its segments will contain at least one. Consequently, repeated sampling of a region—which is what staggering accomplishes over a full scrubbing pass—is more effective than full sequential scrubbing of a region. To see this more clearly, consider an extreme case of clustering: suppose that when a region develops an LSE, all of its segments develop one. In this case, sampling any one segment suffices to detect an LSE-affected region; there is no benefit to scrubbing more than one segment per region. So it is best to sample one segment per region, move on as quickly as possible, and return later to check for fresh LSEs, i.e., to stagger.

Staggering does have a drawback, though. It requires more disk-head movement than sequential scrubbing. (Sequential scrubbing is clearly optimal in terms of disk-head movement.) Thankfully, as we show next, for carefully chosen parameters, the slowdown due to disk-head movement in staggered scrubbing is minimal.

We determined through experiments parameters for the staggered strategy that do not affect performance. The first question we needed to answer is the optimal request size when reading from disk sequentially. As suggested by previous literature [12], read performance improves with increasing request sizes, as function calls and interrupts introduce a performance penalty.

We performed a first experiment in which we read 16GB from a 7200 RPM Hitachi drive using request sizes between 1KB and 64KB. We found that a disk request size of 16KB is nearly optimal; performance improves negligibly for larger request sizes. This suggets that re-

quest sizes in sequential scrubbing strategies should be at least 16KB.

Second, we want to quantify the performance overhead for staggered scrubbing versus sequential reading from disk. We consider staggered scrubbing with regions of different sizes, ranging from 50MB to 500MB, and different request sizes, ranging from 32KB to 2MB. We found out that, while the overhead of staggering for small request sizes (32KB or 64KB) is large (a factor of 5 to 8), the overhead becomes minimal when the request size increases to several MB. For instance, for a request size of 1MB or 2MB, the overhead is about 2%.

These experimental findings provide guidance for our parameter choices in staggered scrubbing. To minimize the performance impact of staggering, we choose a segment size of 1MB. For that segment size, our results show that the staggering overhead is not highly dependent on the region size. We thus choose a region size that aligns with the radius of most error clusters (128MB).

4.3 Strategies with Adaptive Scrubbing Rates

To capture temporal locality of latent sector errors, we introduce scrubbing strategies with scrubbing rates that change adaptively according to drive history. From the results in the NetApp study, we know that monthly LSE rates are fairly constant before the development of the first LSE in a drive. (Again, an exception is the first 60 days of drive operation, which exhibit slightly lower LSE rates.) Once a first LSE develops, i.e., a triggering error, more errors are likely to develop shortly afterward.

We propose to start with a scrubbing rate SR_First60 in the first 60 days of disk operation, and change it to rate SR_PreLSE before any LSEs are detected. Once the disk develops a first LSE, the strategy enters into an *accelerated interval* (with length Int_Acc) and adjusts the scrubbing rate to SR_Acc. At the end of the accelerated interval, the scrubbing rate is modified to SR_PostLSE. The process is repeated every time a LSE is detected: the strategy enters an accelerated interval with an adjusted scrubbing rate, and then reverts to SR_PostLSE. Disks that never develop an LSE are scrubbed with rate SR_First60 in the first 60 days of operation and SR_PreLSE after that.

4.4 Modeling the Design / Search Space of Scrubbing Strategies

Combining the ideas of staggering and adaptive scrubbing rates, we propose an expanded design space of scrubbing strategies that we will search for optimal strategies in the next section of the paper. A strategy in this design space operates as follows. Before the detection of the first LSE, the strategy proceeds in a staggered fashion with scrubbing rates SR_First60 in the first 60 days of drive operation and SR_PreLSE after that. Once a first LSE is detected, the strategy enters into an accelerated interval and switches to a sequential strategy with scrubbing rate SR_Acc. It scrubs sequentially regions of the disk centered at the detected error and continues with regions further away. When the accelerated interval ends, the strategy reverts to staggered scrubbing with rate SR_PostLSE, starting from the first disk sector.

The parameters that characterize our design space are graphically depicted in Figure 4. A point in our design space is given by coordinates (SR_First60, SR_PreLSE, SR_Acc, SR_PostLSE, Int_Acc).

To convert our design space into a search space, i.e., to specify the constraints on our search for optimal strategies, we must choose concrete parameter ranges and granularities. While this is a somewhat heuristic process, experimental guidance motivates the following choices:

- The staggered strategy uses a region of size 128MB, and a segment size of 1MB. These choices were explained in Section 4.2.

- We specify the scrubbing rates in terms of gigabytes scrubbed per hour. We constrain these rates to an interval whose maximum value corresponds to a full disk scrub in one day (which amounts to 20GB/hour for a 500GB disk). We define the search space for these scrubbing rates with a granularity of 0.5GB/hour, starting from the minimum value of 0.5GB/hour.

- The length of interval Int_Acc is a parameter with minimum value 3 hours and maximum value the time it takes to scrub the full disk sequentially with rate SR_Acc. We search this interval at a granularity of 3 hours.

- The size of the regions scrubbed sequentially in accelerated intervals is 128MB, since this is the clustering radius of about 80% of LSEs. We scrub the regions of size 128MB centered at the first error found, and then continue with the regions further away.

5 Simulation Model and Evaluation

Before describing our simulation model, we specify our new metric MLET *(*Mean Latent Error Time). Intuitively, for a single disk with a specified latent error model and scrubbing strategy, MLET measures the average (over LSE patterns) fraction of the total drive operation time during which the drive has undetected LSEs and is thus susceptible to data loss.

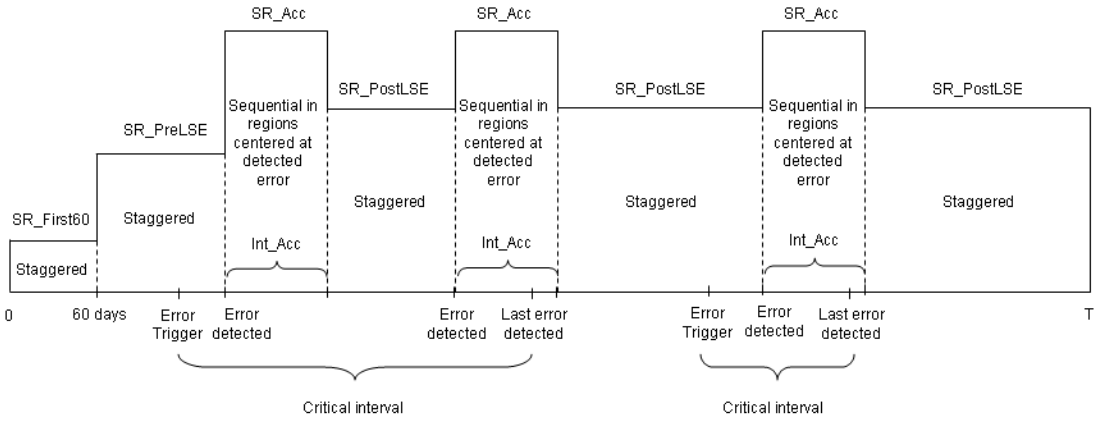


Figure 4: Search space of scrubbing strategies given by parameters SR_First60, SR_PreLSE, SR_Acc, SR_PostLSE and Int_Acc.

Formally, consider a latent sector error probability distribution $\mathcal{P}_{\mathsf{LSE}}$ from space $\mathcal{L}$ and a scrubbing strategy S from space $\mathcal{S}$. For a given pattern of latent-error development LSE from $\mathcal{P}_{\mathsf{LSE}}$, we define the Latent Error Time $\mathrm{LET}(t, \mathsf{LSE}, \mathsf{S})$ as the fraction of the time intervals up to disk age t during which the drive has undetected LSEs. $\mathrm{MLET}(t, \mathsf{S})$ is then defined as the mean of $\mathrm{LET}(t, \mathsf{LSE}, \mathsf{S})$ over the probability distribution $\mathcal{P}_{\mathsf{LSE}}$.

We note that this definition holds for a deterministic scrubbing strategy S. We could extend the definition for probabilistic strategies, to average over the scrubbing strategy distribution $\mathcal{S}$.

5.1 Simulation Model

We have written an event-driven simulation model in Java that simulates the behavior of a disk for T time intervals, each of length one hour. In our experiments, we run our simulation for maximum 24 months for 100,000 disks. (The NetApp data span 24 months of disk operation.) We consider enterprise disk model n-2 and simulate hard drives with a capacity of 500GB. We model the disk normal workload using the HP Cello 99 traces, available from the SNIA IOTTA repository [1]. In our simulation we are interested only in total number of bytes read and written per time interval (i.e., hour). We compute the number of bytes accessed for one hard drive in the original Cello traces. Since these traces are ten years old, we expect that the utilization level is low compared to today's environments. To simulate different utilization levels we scale the number of bytes accessed by a factor between 1 and 100. We simulate both sequential strategies with fixed scrubbing rates and staggered strategies with fixed and adaptive rates.

The events of interest to our simulator are the triggering of age and usage errors, detection of errors, and the moments in time when the scrubbing rate changes, i.e., the disk age reaches 60 days, an accelerated interval begins, or an accelerated interval ends. Age errors are triggered by the distribution derived from the NetApp paper, as described in Section 3.2. The simulator keeps track of the usage rates due to both normal accesses and disk scrubbing and triggers a usage error once the usage for a disk exceeds a random variable normally distributed, as described in Section 3.2.

One important challenge arises in the construction of an efficient simulator. Recall that in our LSE model, a triggering LSE is followed by a cascade of other LSEs. The interval of time between the first error trigger and the detection of all errors in a cluster is what we call a *critical interval*, depicted in Figure 4. It is possible that while in the critical interval of one cluster of errors, another cluster of errors develops. Accommodating a potentially large number of overlapping and nested critical intervals would complicate our model and simulation considerably. For this reason, we make the simplifying assumption that clusters of usage errors do not overlap. We do, however, treat the case in which an age error cluster overlaps with an usage error cluster.

In practice, following a LSE detection, a logical-to-physical remapping of the affected sector takes place. We do not consider the effect of this remapping in our simu-

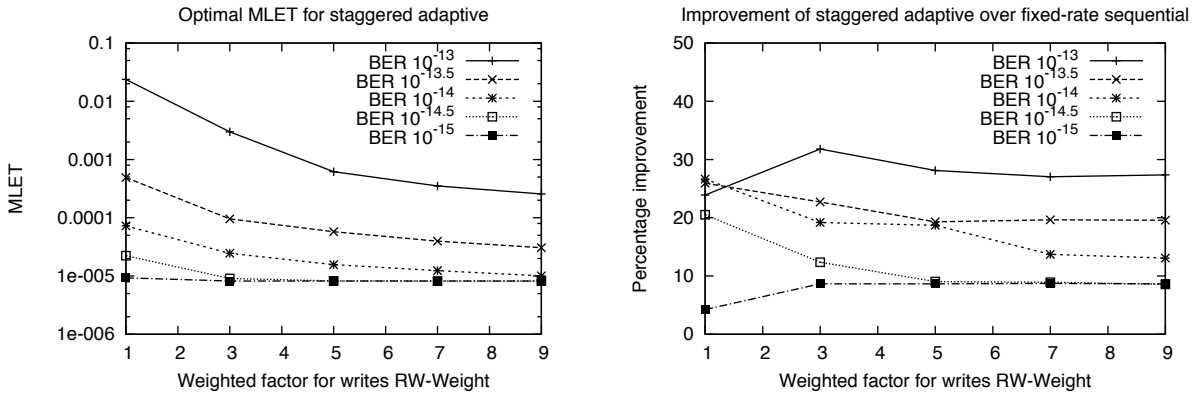


Figure 5: Optimal MLET for staggered adaptive strategies (left) and its relative percentage improvement compared to optimal fixed rate sequential strategies (right) as a function of different BERs and weighted factors for write.

lation model, but this needs to be addressed in an actual implementation of scrubbing strategies in hard drives.

5.2 Simulation Results

Our goal is to determine optimal scrubbing strategies in the design space outlined in Section 4.4. Since our design space for scrubbing strategies proved to be too large to be searched exhaustively in an efficient manner, we implemented a more efficient heuristic search algorithm. Based on brief experimentation, we believe that this heuristic finds strategies close to optimal. For a fixed BER, read/write weight RW_Weight, and disk workload, the algorithm to determine an approximation to the optimal scrubbing strategy in our design space is the following:

- We search exhaustively for the scrub rate λ (between 0.5GB/hour and maximum scrubbing rate) that achieves the minimum MLET for staggered fixed-rate strategies.

- We vary the rate in the accelerated interval between λ and the maximum scrub rate (given by a full scrub per day), and the length of the accelerated interval (between 3 hours and the time it takes to scrub the full disk with the accelerated scrub rate). We determine thus the scrub rate λ_{acc} and the length of accelerated interval $\mathsf{int\_acc}$ that minimize MLET.

- We vary the rate in the first 60 days from 0.5GB/hour to the maximum allowed scrub rate, and determine λ_{60} that minimizes MLET. Similarly, we vary SR_PreLSE and SR_PostLSE to determine $\lambda_{\mathsf{prelse}}$ and $\lambda_{\mathsf{postlse}}$.

- We output the point $(\lambda_{60}, \lambda_{\mathsf{prelse}}, \lambda_{\mathsf{acc}}, \lambda_{\mathsf{postlse}}, \mathsf{int\_acc})$ as an estimate of the optimal strategy.

In the rest of the paper, we sometimes refer to the output of the previous algorithm as "optimal strategy".

Optimal strategy dependence on different BER and read/write weights. First, we show how the optimal scrubbing strategy depends on the drive BER and the read/write weight RW_Weight. We plot on the left graph in Figure 5 the optimal MLET for staggered adaptive strategies and on the right graph in Figure 5 its relative improvement compared to optimal fixed-rate sequential strategies. We vary BER between 10^{-15} and 10^{-13}, and the read/write weight between 1 (i.e., read and write contribute equally to disk wear-out) and 9 (i.e., contribution of reads to disk wear-out is 9 times lower than that of writes).

The left graph in Figure 5 shows how MLET decreases for more reliable disks (i.e., disks with higher BER): for instance, for a read/write weight of 1, MLET varies between 0.031 for a 10^{-13} BER to $9.69 \cdot 10^{-5}$ for a 10^{-15} BER. As expected, MLET also decreases when the disk wear-out due to reads is lower (i.e., the read/write weight increases), as the disk is developing fewer usage errors.

From the right graph in Figure 5, we infer that the staggered adaptive strategy improves MLET relative to the optimal fixed-rate sequential strategy by at most 30%. Improvements are larger for disks with higher development of usage errors. We expect that this effect will be amplified when considering RAID-5 or RAID-6 configurations with multiple disks. In RAID-5, for instance, data loss occurs when a drive failure is coupled with a latent error on any of the other drives. The vulnerability interval due to latent errors (the time intervals in which at least one drive has undetected LSEs) consists of all vulnerability intervals of the drives in the RAID configuration. Consequently, a reduction in the MLET metric for one drive will produce an amplified reduction on the length of the vulnerability interval for the array (roughly

BER		Weighted factor for writes				
		1	3	5	7	9
10^{-13}	fixed-rate	4	0.5	0.5	0.5	1
	adaptive	(0.5,10,18.5,2.5)	(0.5,12.5,14.5,0.5)	(0.5,0.5,12.5,0.5)	(0.5,0.5,18.5,0.5)	(0.5,1,17,1.5)
$10^{-13.5}$	fixed-rate	0.5	1.5	2.5	4	5
	adaptive	(0.5,0.5,12.5,0.5)	(1,1.5,15.5,1.5)	(3,3,14,3)	(3.5,3.5,17,3.5)	(5,5,18,5)
10^{-14}	fixed-rate	2	6	9.5	12.5	17.5
	adaptive	(1,2,18,1)	(2,6,19.5,5)	(10,10,18.5,10)	(13,13,19,13)	(18,18,19.5,18)
$10^{-14.5}$	fixed-rate	6.5	19	20	20	20
	adaptive	(7,7,19,7)	(12.5,20,20,20)	(17,20,20,20)	(17,20,20,20)	(17,20,20,20)
10^{-15}	fixed-rate	20	20	20	20	20
	adaptive	(19,19,19,19)	(17,20,20,20)	(17,20,20,20)	(17,20,20,20)	(17,20,20,20)

Table 4: Optimal points for sequential fixed-rate and adaptive staggered strategies for different BERs and weighted factors for writes. For sequential fixed-rate strategy, the table includes the optimal scrubbing rate. For the adaptive staggered strategy, the table shows the optimal point (SR_First60, SR_PreLSE, SR_Acc, SR_PostLSE).

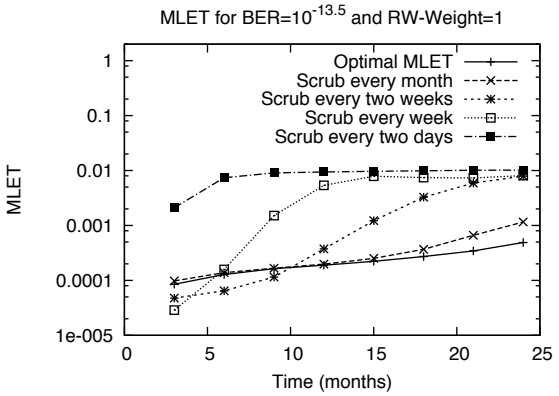


Figure 6: MLET for optimal strategy and several sequential strategies for disks with high usage errors.

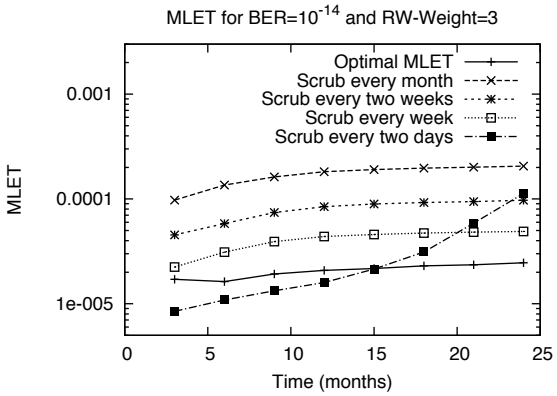


Figure 7: MLET for optimal strategy and several sequential strategies for disks with medium usage errors.

scaled by the number of drives in the RAID configuration).

Table 4 gives an interesting insight on the optimal scrubbing rates used by both fixed-rate sequential and adaptive staggered strategies. For disks featuring high development of usage errors (due to high BER, and low read/write weight), the optimal fixed-rate sequential strategy is using a fairly low scrubbing rate (since in this case the scrubbing process itself will contribute to disk wear-out and LSE development). The optimal staggered adaptive strategy also uses low scrub rates, except for accelerated intervals, when the scrubbing rate is increased to almost maximum allowed rate to detect LSEs quickly. In contrast, for disks developing few usage errors (due to low BER and high read/write weight), the optimal scrubbing strategies (both sequential and staggered adaptive) use a high scrubbing rate that is close to the maximum allowed rate.

Improvement of staggered adaptive strategy over several widely used fixed-rate sequential scrubbing strategies. We compare next the MLET metric for the optimal adaptive staggered strategy and various fixed-rate sequential strategies (i.e., scrub the disk once a month, once every two weeks, once every week, and once every two days). These fixed-rate sequential strategies are widely used today in many systems. Graphs in Figures 6, 7 and 8 show the MLET metric for these strategies as a function of the simulation interval. The results demonstrate that by using more intelligent scrubbing than the ad-hoc approaches in use today, the MLET metric can be improved by at least a factor of two and at most a factor of 20.

An important observation derived from these graphs is that optimal strategies are highly dependent on disk characteristics. For disks that develop a high number of usage errors (Figure 6 with BER $10^{-13.5}$ and the read/write weight 1), the optimal adaptive staggered strategy is clos-

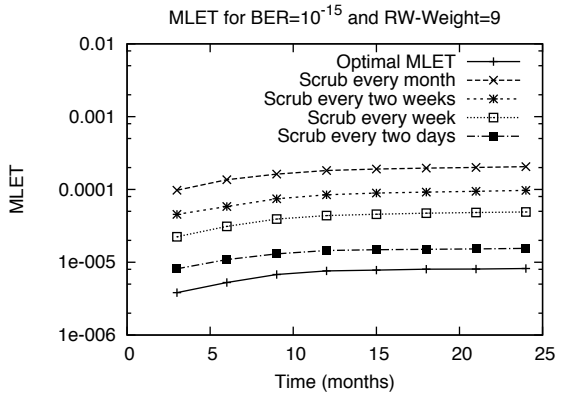


Figure 8: MLET for optimal strategy and several sequential strategies for disks with low usage errors.

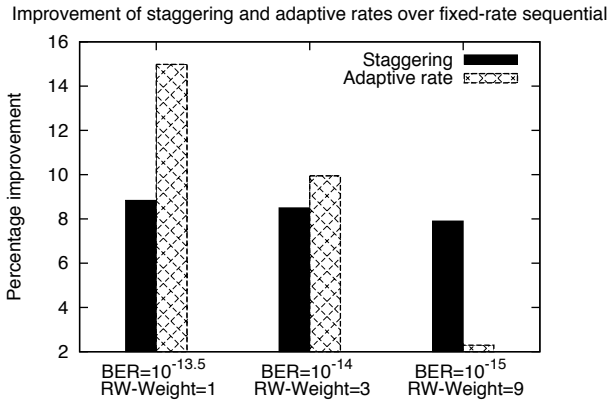


Figure 9: Relative improvement in MLET for staggering and adaptive rates compared to fixed-rate sequential.

est to scrubbing the disk once every month (i.e., infrequent scrubbing). For disks with medium number of usage errors (Figure 7 with BER 10^{-14} and the read/write weight 3), the optimal strategy is closer to scrubbing the disk once every week. In Figure 8, disks that develop low number of usage errors (e.g., BER 10^{-15} and the read/write weight 9) have optimal strategies closer to scrubbing every two days. This clearly demonstrates that it is infeasible to develop a good "one-size-fit-all" recipe for disk scrubbing.

Interestingly, Figures 6 and 7 show that the optimal strategy for time t is not always the optimal strategy for all previous time intervals. This observation suggests that we could achieve further optimizations when designing scrubbing strategies by expanding our search space. In particular, an idea that deserves further exploration is to periodically adapt the scrubbing strategy over time. Instead of computing one optimal strategy for the entire drive operational time, we could instead compute new optimal strategies for short time intervals (e.g., 3 or 6 months). With this approach, the optimal strategy for disks that develop a medium number of errors, for instance, is to scrub with a constant rate (once every two weeks) for the first 15 months, and then switch to an adaptive staggered strategy.

Benefit of staggered and adaptive strategies. We assess next the benefit of our two main optimizations: using a staggered approach for scrubbing, and varying scrubbing rates adaptively. We show in Figure 9 relative improvements of these two optimizations compared to the optimal fixed-rate sequential strategy. We plot results for disks with three different characteristics, classified by the occurrence of high, medium or low occurrence of usage errors, respectively.

We observe that the idea of staggering compared to sequentially reading the disk produces a steady improvement in MLET by around 10% for all disk characteristics. On the other hand, adaptively changing the scrubbing rate has a greater impact on disks that develop a higher number of usage errors. The relative improvement in MLET by adaptively changing the scrubbing rate is as high as 15% for disks with a high number of usage errors, and as low as 2% for most reliable disks. These results are consistent with our previous observation that the optimal scrubbing strategy for disks with few usage errors is scrubbing at the maximum fixed rate.

Interestingly, a paper concurrently and independently written [13] shows that our experimental results might underestimate the benefit of the staggering technique. Schroeder et al. [13] evaluate staggered scrubbing in comparison with fixed-rate sequential strategies on real failure data and report that staggered scrubbing can improve mean time of error detection compared to sequential scrubbing by up to 40%. While Schroeder et al. use a different metric in comparing different scrubbing strategies, these results confirm the benefit of staggering.

Optimal strategy dependence on disk workloads. Finally, we assess the impact of different disk workloads on optimal scrubbing strategies. We consider the workloads of one disk from the HP Cello 1999 I/O traces, and scale them by a factor of 1, 10 and 100. We plot on the left of Figure 10 the MLET value for optimal staggered adaptive strategy and on the right its relative improvement compared to fixed-rate sequential strategies.

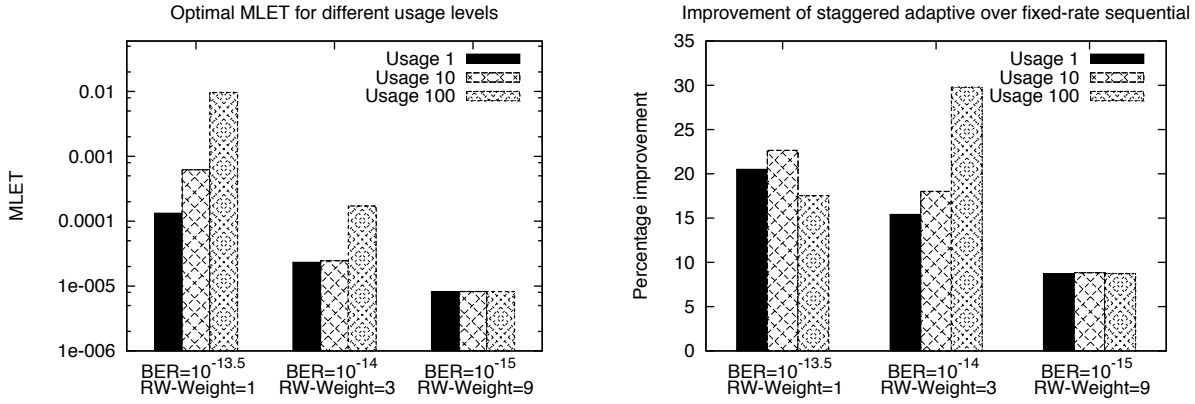


Figure 10: Optimal MLET for staggered adaptive strategies (left) and its relative percentage improvement compared to optimal fixed rate sequential strategies (right) for different disk characteristics and different workloads.

In both graphs, usage levels are scaled by a factor of 1, 10 and 100, respectively. As in previous experiments, we consider disks that develop a high, medium and low level of usage errors.

The left graph in Figure 10 shows that disks developing high and medium number of usage errors exhibit sensitivity to normal access workloads. In particular, scaling the disk workloads by a factor of 10 has the effect of increasing the optimal MLET metric by an order of magnitude for disks developing a high number of usage errors. Disks that exhibit low number of usage errors are not sensitive to disk workloads at all.

The right graph in Figure 10 shows the relative improvement of the optimal staggered adaptive strategy compared to the optimal fixed-rate sequential strategy for different disk usage levels. Disks exhibiting high and medium development of usage errors benefit mostly from the staggered adaptive technique. For these types of disks, the relative improvements of the staggered adaptive strategy increase with higher disk utilization. The exception is the case of disks developing high number of usage errors under heavy workload (scaled by a factor of 100). In that case, we conjecture that the number of usage errors increases greatly, leading to lower relative improvements of the staggered adaptive strategy than for lower disk utilization. We observe again that disks developing a low number of errors are insensitive to disk workloads: the relative improvement of the staggered adaptive strategy is around 10%, independent of the disk workload.

Discussion. We have demonstrated that we can design more intelligent scrubbing algorithms than those in use today by taking into account disk characteristics and the history of error development. We have characterized the resilience of a single drive to latent sector errors by defining the new MLET metric. Our results demonstrate that optimal scrubbing strategies need to be carefully crafted for different disk characteristics. In particular, optimal strategies are highly dependent on the BER and the read/write weight RW_Weight of a disk.

For disks that develop a high number of usage errors, scrubbing benefits greatly from adaptively changing rates. The optimal strategy uses a low scrubbing rate, that is increased to almost the maximum allowed rate in the accelerated interval immediately following the detection of a LSE. For disks that develop a low number of usage errors, the optimal strategy uses the maximum allowed scrubbing rate that does not interfere with the normal disk usage. Staggering across disk regions instead of sequentially reading the disk improves the MLET metric for all disk models.

Our optimal scrubbing strategies can improve the MLET metric compared to widely used strategies (e.g., scrub the disk sequentially once every week) by an order of magnitude. We expect that this effect will be amplified when considering the MTTDL metric for an array of disks (e.g., RAID-5 or RAID-6 configuration).

A limitation of the current work is the high sensitivity of the results to disk parameters that are not always made public by disk manufacturers. We hope that, as more failure data becomes available, our results can be further refined by the community.

6 Conclusions

Our work is a first step in the exploration of more intelligent scrubbing strategies for hard drives. It shows that single drive reliability can be greatly improved by expanding the design space for scrubbing strategies beyond naïve sequential and constant-rate approaches.

Several challenging options for further research arise in our work. The first is an expansion of our design and search spaces for scrubbing strategies. Appealing to search heuristics such as hillclimbing or simulated annealing would enable us to consider a more fine-grained and sophisticated design space.

Second, we plan to evaluate the performance overhead of various scrubbing strategies in conjunction with realistic disk workloads.

Third, with the emergence of FLASH technology, an intriguing question is how (and if) our results translate into the FLASH realm. With completely different physical characteristics than hard drives, and a complex physical-to-logical translation layer, FLASH would seem a challenging target for the development of latent error and scrubbing models.

Finally, we have only studied the effect of scrubbing on single-drive reliability. Extension of our work to a systemic analysis in the context of replication systems like RAID seems an interesting area of future research.

7 Acknowledgements

We thank Ron Rivest, Burt Kaliski and Kevin Bowers for numerous insightful discussions during the progress of this work. We also thank Bianca Schroeder and Georgios Amvrosiadis for conversations on latent error modeling and staggering strategies. Finally, we would like to extend our gratitude to our shepherd, Jim Plank, and the anonymous reviewers for their careful suggestions in revising the final version of the paper.

References

[1] The SNIA IOTTA Repository. `http://iotta.snia.org/`.

[2] L.N. Bairavasundaram, G.R. Goodson, S. Pasupathy, and J. Schindler. An analysis of latent sector errors in disk drives. In *ACM SIGMETRICS*, pages 289—300, 2007.

[3] M. Baker, M. A. Shah, D. S. H. Rosenthal, M. Roussopoulos, P. Maniatis, T. J. Giuli, and P. P. Bungale. A fresh look at the reliability of long-term digital storage. In *1st ACM SIGOPS/EuroSys*, pages 221—234, 2006.

[4] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. Raid: high-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185.

[5] A. Dholakia, E. Eleftheriou, X. Hu, I. Iliadis, J Menon, and K. K. Rao. A new intra-disk redundancy scheme for high-reliability RAID storage systems in the presence of unrecoverable errors. *ACM Transactions on Storage*, 4(1), 2008.

[6] J.G. Elerath and M. Pecht. Enhanced reliability modeling of RAID storage systems. In *37th Annual IEEE/IFIP DSN*, pages 175—184, 2007.

[7] I. Iliadis, R. Haas, X. Y. Hu, and E. Eleftheriou. Disk scrubbing versus intra-disk redundancy for high-reliability RAID storage systems. In *ACM SIGMETRICS*, pages 241—252, 2008.

[8] H. Kari. *Latent Sector Faults and Reliability of Disk Arrays*. PhD thesis, Helsinki University of Technology, 1997.

[9] N. Mi, A. Riska, E. Smirni, and E. Riedel. Enhancing data availability through background activities. In *38th Annual IEEE/IFIP DSN*, 2008.

[10] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD*, pages 109—116, 1988.

[11] E. Pinheiro, W. D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *5th USENIX FAST*, 2007.

[12] E. Riedel, C. van Ingen, and J. Gray. A performance study of sequential I/O on windows NT. In *Second USENIX Windows NT Symposium*, 1998.

[13] B. Schroeder, S. Damouras, and P. Gill. Understanding latent sector errors and how to protect against them. In *8th USENIX FAST*, 2010.

[14] B. Schroeder and G. Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean too you? In *5th USENIX FAST*, 2007.

[15] T. Schwarz, Q. Xin, E. L. Miller, D. D. E. Long, A. Hospodor, and S. Ng. Disk scrubbing in large archival storage systems. In *IEEE 12th MASCOTS*, 2004.

Understanding latent sector errors and how to protect against them

Bianca Schroeder
Dept. of Computer Science
University of Toronto
Toronto, Canada
bianca@cs.toronto.edu

Sotirios Damouras
Dept. of Statistics
University of Toronto
Toronto, Canada
sotirios@utstat.toronto.edu

Phillipa Gill
Dept. of Computer Science
University of Toronto
Toronto, Canada
phillipa@cs.toronto.edu

Abstract

Latent sector errors (LSEs) refer to the situation where particular sectors on a drive become inaccessible. LSEs are a critical factor in data reliability, since a single LSE can lead to data loss when encountered during RAID reconstruction after a disk failure. LSEs happen at a significant rate in the field [1], and are expected to grow more frequent with new drive technologies and increasing drive capacities. While two approaches, data scrubbing and intra-disk redundancy, have been proposed to reduce data loss due to LSEs, none of these approaches has been evaluated on real field data.

This paper makes two contributions. We provide an extended statistical analysis of latent sector errors in the field, specifically from the view point of how to protect against LSEs. In addition to providing interesting insights into LSEs, we hope the results (including parameters for models we fit to the data) will help researchers and practitioners without access to data in driving their simulations or analysis of LSEs. Our second contribution is an evaluation of five different scrubbing policies and five different intra-disk redundancy schemes and their potential in protecting against LSEs. Our study includes schemes and policies that have been suggested before, but have never been evaluated on field data, as well as new policies that we propose based on our analysis of LSEs in the field.

1 Motivation

Over the past decades many techniques have been proposed to protect against data loss due to hard disk failures [3, 4, 8, 9, 14, 15, 18]. While early work focused on total disk failures, new drive technologies and increasing capacities have led to new failure modes. A particular concern are *latent sector errors (LSEs)*, where individual sectors on a drive become unavailable. LSEs are caused, for example, by write errors (such as a high-fly write) or by media imperfections, like scratches or smeared soft particles.

There are several reasons for the recent shift of attention to LSEs as a critical factor in data reliability. First and most importantly, a single LSE can cause data loss when encountered during RAID reconstruction after a disk failure. Secondly, with multi-terabyte drives using perpendicular recording hitting the markets, the frequency of LSEs is expected to increase, due to higher areal densities, narrower track widths, lower flying heads, and susceptibility to scratching by softer particle contaminants [6]. Finally, LSEs are a particularly insidious failure mode, since these errors are not detected until the affected sector is accessed.

The mechanism most commonly used in practice to protect against LSEs is a background scrubber [2, 12, 13, 17] that continually scans the disk during idle periods in order to proactively detect LSEs and then correct them using RAID redundancy. Several commercial storage systems employ a background scrubber, including, for example, NetApp's systems.

Another mechanism for protection against LSEs is intra-disk redundancy, i.e. an additional level of redundancy inside each disk, in addition to the inter-disk redundancy provided by RAID. Dholakia et al. [5, 10] recently suggested that intra-disk redundancy can make a system as reliable as a system without LSEs.

Devising effective new protection mechanisms or obtaining a realistic understanding of the effectiveness of existing mechanisms requires a detailed understanding of the properties of LSEs. To this point, there exists only one large-scale field study of LSEs [1], and no field data that is publicly available. As a result, existing work typically relies on hypothetical assumptions, such as LSEs that follow a Poisson process [2, 7, 10, 17]. None of the approaches described above for protecting against LSEs has been evaluated on field data.

This paper provides two main contributions. The first contribution is an extended statistical study of the data

in [1]. While [1] provides a general analysis of the data, we focus in our study on a specific set of questions that are relevant from the point of view of how to protect against data loss due to LSEs. We hope that this analysis will help practitioners in the field, who operate large-scale storage systems and need to understand LSEs, as well as researchers who want to simulate or analyze systems with LSEs and don't have access to field data. It will also give us some initial intuition on the real-world potential of different protection schemes that have been proposed and what other schemes might work well.

The second contribution is an evaluation of different approaches for protecting against LSEs, using the field data from [1]. Our study includes several intra-disk redundancy schemes (simple parity check schemes, interleaved parity [5, 10], maximum distance separable erasure codes, and two new policies that we propose) and several scrubbing policies, including standard sequential scrubbing, the recently proposed staggered scrubbing [13] and some new policies.

The paper is organized as follows. We provide some background information on LSEs and the data we are using in Section 2. Section 3 presents a statistical analysis of the data. Section 4 evaluates the effectiveness of intra-disk redundancy for protecting against LSEs and Section 5 evaluates the effectiveness of proactive error detection through scrubbing. We discuss the implications of our results in Section 6.

2 Background and Data

For our study, we obtained a subset of the data that was used by Bairavasundaram et al. [1]. While we refer the reader to [1] for a full description of the data, the systems they come from and the error handling mechanisms in those systems, we provide a brief summary below.

Bairavasundaram et al. collected data on disk errors on NetApp production storage systems installed at customer sites over a period of 32 months. These systems implement a proprietary software stack consisting of the WAFL filesystem, a RAID layer and the storage layer. The handling of latent sector errors in these systems depends on the type of disk request that encounters an erroneous sector and the type of disk. For enterprise class disks, the storage layer re-maps the disk request to another (spare) sector. For read operations, the RAID layer needs to reconstruct the data before the storage layer can remap it. For nearline disks, the process for reads is similar, however the remapping of failed writes is performed internally by the disk and transparent to the storage layer. All systems periodically scrub their disks to proactively detect LSEs. The scrub is performed using the SCSI verify command, which validates a sector's integrity without transferring data to the storage layer. A typical scrub interval is 2 weeks. Bairavasundaram et al. found that the majority of the LSEs in their study (more than 60%) were detected by the scrubber, rather than an application access.

In total the collected data covers more than 1.5 million drives and contains information on three different types of disk errors: latent sector errors, not-ready-condition-errors and recovered errors. Bairavasundaram et al. find that a significant fraction of drives (3.45%) develops latent sector errors at some point in their life and that the fraction of drives affected by LSEs grows as disk capacity increases. They also study some of the temporal and spatial dependencies between errors and find evidence of correlations between the three different types of errors.

For our work, we have been able to obtain a subset of the data used in [1]. This subset is limited to information on latent sector errors (no information on not-ready-condition-errors and recovered errors) and contains for each drive that developed LSEs information on the time when the error was detected and the logical block number of the sector that was affected. Note that since LSEs are by definition *latent* errors, i.e. errors that are unknown to the system until it tries to access the affected sector, we cannot know for sure when exactly the error happened. The timestamps in our data refer to the time when the error was detected, not necessarily when it first happened. We can, however, narrow down the time of occurrence to a 2-week time window: since the scrub interval in NetApp's systems is two weeks, any error must have happened within less than two weeks before the detection time. For applications in this paper where the timestamp of an error matters we use three different methods for approximating timestamps, based on the above observation, in addition to using the timestamps directly from the trace. We describe the details in Section 5.1.

We focus in our study on drives that have been in the field for at least 12 months and have experienced at least one LSE. We concentrate on the four most common nearline drive models (the models referred to as A-1, D-2, E-1, E-2 in [1]) and the four most common enterprise drive models (k-2, k-3, n-3, and o-3). In total, the data covers 29,615 nearline drives and 17,513 enterprise drives.

3 Statistical properties of LSEs

We begin with a study of several statistical properties of LSEs. Many baseline statistics, such as the frequency of LSEs and basic temporal and spatial properties, have been covered by Bairavasundaram et al. in [1], and we are not repeating them here. Instead we focus on a specific set of questions that is relevant from the point of view of how to protect against data loss due to LSEs.

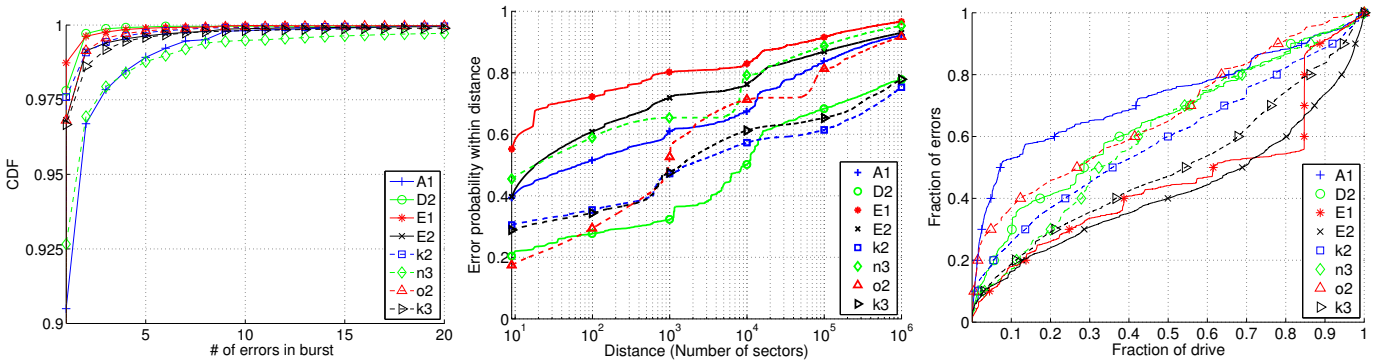


Figure 1: *Distribution of the number of contiguous errors in a burst (left), cumulative distribution function of the sector distance between errors that occur within same 2-week interval (middle), and the location of errors on the drive (right)*

3.1 How long are error bursts?

When trying to protect against LSEs, it is important to understand the distribution of the lengths of error bursts. By an error burst we mean a series of errors that is contiguous in logical block space. The effectiveness of intra-disk redundancy schemes, for example, depends on the length of bursts, as a large number of contiguous errors likely affects multiple sectors in the same parity group preventing recovery through intra-disk redundancy.

Figure 1(left) shows for each model the cumulative distribution function of the length of error bursts. We observe that in 90–98% of cases a burst consists of one single error. For all models, except A-1 and n-3, less than 2.5% of runs consist of two errors and less than 2.5% have more than 2 errors.

An interesting question is how to best model the length of an error burst and the number of good sectors that separate two bursts. The most commonly used model is a geometric distribution, as it is convenient to use and easy to analyze. We experimented with 5 different distributions (Geometric, Weibull, Rayleigh, Pareto, and Lognormal), that are commonly used in the context of system reliability, and evaluated their fit through the total squared differences between the actual and hypothesized frequencies (χ^2 statistic). We found consistently across all models that the geometric distribution is a poor fit, while the Pareto distribution provides the best fit. For the length of the error bursts, the deviation of the geometric from the empirical distribution was more than 13 times higher than that of the Pareto (13.50 for nearline and 14.34 for enterprise), as measured by the χ^2 statistic. For the distance between bursts the geometric fit was even worse. The deviation under the geometric distribution compared to the Pareto distribution is 46 and 110 times higher for nearline and enterprise disks, respectively. The geometric distribution proved such a poor fit because it failed to capture the long tail behavior of the data, i.e. the presence of long error bursts and the clustering of errors.

The top two rows in Table 1 summarize the parameters for the Pareto distribution that provided the best fit. For the number of good sectors between error bursts the parameter in the table is the α parameter of the Pareto distribution. For modeling the burst lengths we used two parameters. The first parameter p gives the probability that the burst consists of a single error, i.e. $(1-p)$ is the probability that an error burst will be longer than one error. The second parameter is the α parameter of the Pareto distribution that best fits the number of errors in bursts of length > 1.

3.2 How far are errors spaced apart?

Knowing at what distances errors are typically spaced apart is relevant for both scrubbing and intra-disk redundancy. For example, errors that are close together in space are likely to affect several sectors in the same parity group of an intra-disk redundancy scheme. If they also happen close together in time it is unlikely that the system has recovered the first error before the second error happened.

Figure 1 (middle) shows the cumulative distribution function (CDF) of the distance between an error and the closest neighbor that was detected within a 2-week period (provided that there was another error within 2 weeks from the first). We chose a period of 2 weeks, since this is the typical scrub interval in NetApp's filers.

Not surprisingly we find that very small distances are the most common. Between 20–60% of all errors have a neighbor within a distance of less than 10 sectors in logical sector space. However, we also observe that almost all models have pronounced "bumps" (parts where the CDF is steeper) indicating higher probability mass in these areas. For example, model o-2 has bumps at distances of around 10^3 and 10^5 sectors. Interestingly, we also observe that the regions where bumps occur tend

Variable	Dist./Params.	A-1	D-2	E-1	E-2	k-2	k-3	n-3	o-2
Error burst length	Pareto p, α	0.9, 1.21	0.98, 1.79	0.98, 1.35	0.96, 1.17	0.97, 1.2	0.97, 1.15	0.93, 1.25	0.97, 1.44
Distance btw. bursts	Pareto α	0.008	0.022	0.158	0.128	0.017	0.00045	0.077	0.05
#LSEs in 2 weeks	Pareto α	0.73	0.93	0.63	0.82	0.80	0.70	0.45	0.22
#LSEs per drive	Pareto α	0.58	0.81	0.34	0.44	0.63	0.58	0.31	0.11

Table 1: *Parameters from distribution fitting*

to be consistent for different models of the same family. For example, the CDFs of models E-1 and E-2 follow a similar shape, as do the CDFs for models k-2 and k-3. We therefore speculate that some of these distances with higher probability are related to the disk geometry of a model, such as the number of sectors on a track.

3.3 Where on the drive are errors located?

The next question we ask is whether certain parts of the drive are more likely to develop errors than others. Understanding the answer to this question might help in devising smarter scrubbing or redundancy schemes that employ stronger protection mechanisms (e.g. more frequent scrubbing or stronger erasure codes) for those parts of the drive that are more likely to develop errors.

Figure 1 (right) shows the CDF of the logical sector numbers with errors. Note that the X-axis does not contain absolute sector numbers, since this would reveal the capacity of the different models, information that is considered confidential. Instead, the X-axis shows percentage of the logical sector space, i.e. the point (x,y) in the graph means that y% of all errors happened in the first x% of the logical sector space.

We make two interesting observations: The first part of the drive shows a clearly higher concentration of errors than the remainder of the drive. Depending on the model, between 20% and 50% of all errors are located in the first 10% of the drive's logical sector space. Similarly, for some models the end of the drive has a higher concentration. For models E-2 and k-3, 30% and 20% of all errors, respectively, are concentrated in the highest 10% of the logical sector space. The second observation is that some models show three or four "bumps" in the distribution that are equidistant in logical sector space (e.g. model A-1 has bumps at fractions of around 0.1, 0.4 and 0.7 of the logical sector space).

We speculate the areas of the drive with an increased concentration of errors might be are areas with different usage patterns, e.g. filesystems often store metadata at the beginning of the drive.

3.4 What is the burstiness of errors in time?

While Bairavasundaram et al. [1] provide general evidence of temporal locality between errors, the specific question we are interested in here is how quickly exactly the probability of seeing another error drops off with time and how errors are distributed over time. Understanding the conditional probability of seeing an error in a month, given that there was an error x months ago, is useful for scrubbing policies that want to adapt the scrubbing rate as a function of the current probability of seeing an error.

To answer the question above, Figure 2 (left) considers for each drive the time of the first error and shows for each subsequent 2-week period the probability of seeing an additional error. We chose 2-week intervals, since this is the typical scrubbing interval in NetApp's systems, and hence the resolution of the error detection time. We observe that after the first month after the first error is detected, the probability of seeing additional errors drops off exponentially (note the log-scale on the Y-axis), dropping close to 1% after only 10 weeks and below 0.1% after 30 weeks.

Figure 2 (middle) illustrates how errors are distributed over time. We observe each drive for one year after its first error and count how many 2-week scrub intervals in this time period encounter any errors. We observe that for 55–85% of drives, all errors are concentrated in the same 2-week period. Only 10–15% of drives experience errors in two different 2-week periods, and for most models less than 15% see errors in more than two 2-week periods.

Summarizing the above observations, we find that the errors a drive experiences occur in a few short bursts, i.e. errors are highly concentrated in a few short time intervals. One might suspect that this bursty behavior is poorly modeled by a Poisson process, which is often used in modeling LSE arrivals [2, 7, 10, 17]. The reason for the common use of Poisson processes in modeling LSEs is that they are easy to analyze and that so far little data has been available that allows the creation of more realistic models. We fitted a Poisson distribution to the number of errors observed in a 2-week time interval and to the number of errors a drive experiences during its lifetime, and found the Poisson distribution to be a poor fit in both cases. We observe that the empirical distribution has a significantly longer tail than a Poisson distribution, and find that instead a Pareto distribution is a much better fit. For illustration, Figure 2 (right) shows for model n-3 the empirical distribution for the number of errors in a disks's lifetime and the Poisson and Pareto distributions

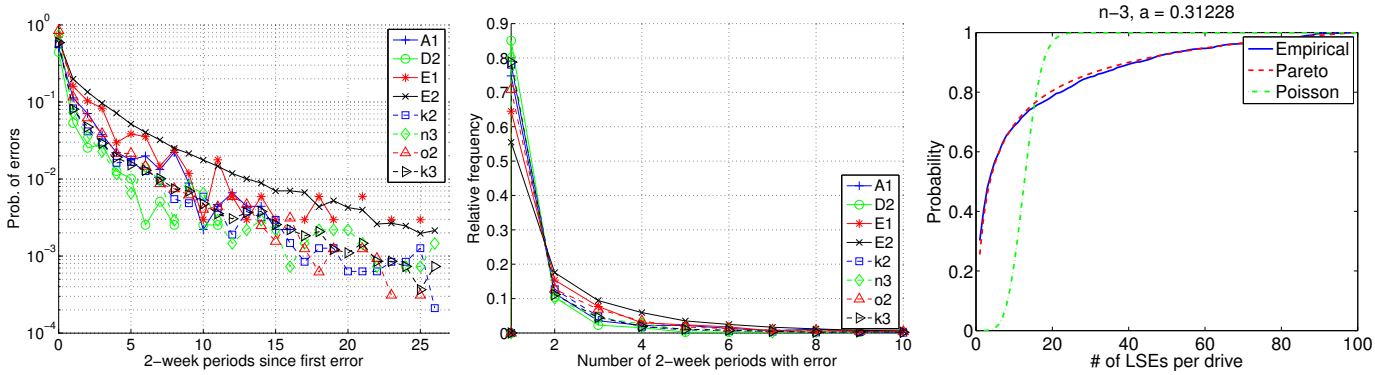


Figure 2: *The probability of seeing an error x 2-week periods after first error (left), the number of 2-week periods in a disk's life with at least one error (middle), and the distribution of the number of errors a disk sees in its lifetime (right).*

fitted to it. We provide the Pareto α parameter for both empirical distributions for all models in Table 1.

3.5 What causes LSEs?

This is obviously a broad question that we cannot hope to answer with the data we have. Nevertheless, we want to address this question briefly, since our observations in Section 3.3 might lead to hasty conclusions. In particular, a possible explanation for the concentration of errors in certain parts of the drive might be that these areas see a higher utilization. While we do not have access to workload data for NetApp's systems, we have been able to obtain two years of data on workload, environmental factors and LSE rates for five large ($> 50,000$ drives each) clusters at Google containing five different drive models. None of the clusters showed a correlation between either the number of reads or the number of writes that a drive sees (as reported by the drive's SMART parameters) and the number of LSEs it develops. We plan a detailed study of workload and environmental factors and how they impact LSEs as part of future work.

3.6 Does close in space mean close in time?

Prior work [1] and the questions above have focused on spatial and temporal correlations in isolation. For most error protection schemes, it is crucial to understand the relationship between temporal and spatial correlation. For example, for intra-disk redundancy schemes it does not only matter how long a burst of errors is (i.e. the number of consecutive errors in the burst), but also how much time there is between errors in a burst. More time between errors increases the chance that the first error is detected and corrected before the second error happens.

Figure 3 (left) shows the distribution of the time an error burst spans, i.e. the time difference between the first and last error in a burst. We observe that in more than 90% of the bursts the errors are discovered within the same 2-week scrub interval and in more than 95% of bursts the errors are detected within a month from each other. Less than 2% of error bursts span more than 3 months. These observations indicate that the errors in most bursts are likely caused by the same event and hence occurred at the same time.

Figure 3 (right) shows a more general view of the correlation between spatial and temporal locality. The graph shows for radii ranging from one sector to 50GB two bars: the first gives the probability that an error has at least one neighbor within this radius at some point during the disk's lifetime; the second bar gives the probability that an error has at least one neighbor within this radius within 2 weeks of time. As the graph shows, for small radii the two bars are virtually identical, indicating that errors that happened close in space were likely caused by the same event and hence happened at nearly the same time. We also observe that even for larger radii the two bars are still very close to each other. The figure shows results for model n-3, but we found results to be similar for all other models.

4 Protecting against LSEs with Intra-disk Redundancy

While inter-disk redundancy has a long history [3, 4, 8, 9, 14, 14, 15, 18], there are much fewer instances of intra-disk redundancy. Some filesystems [11] create in-disk replicas of selected metadata, IRON file systems [16] suggest to add a parity block per file, and recent work by Dholakia et al. [5, 10] introduces a new intra-disk redundancy scheme for all data blocks in a drive.

The motivation behind intra-disk redundancy is to reduce data loss when LSEs are encountered during RAID reconstruction, or where there is no inter-disk redundancy available. Dholakia et al. [5, 10] predict that with

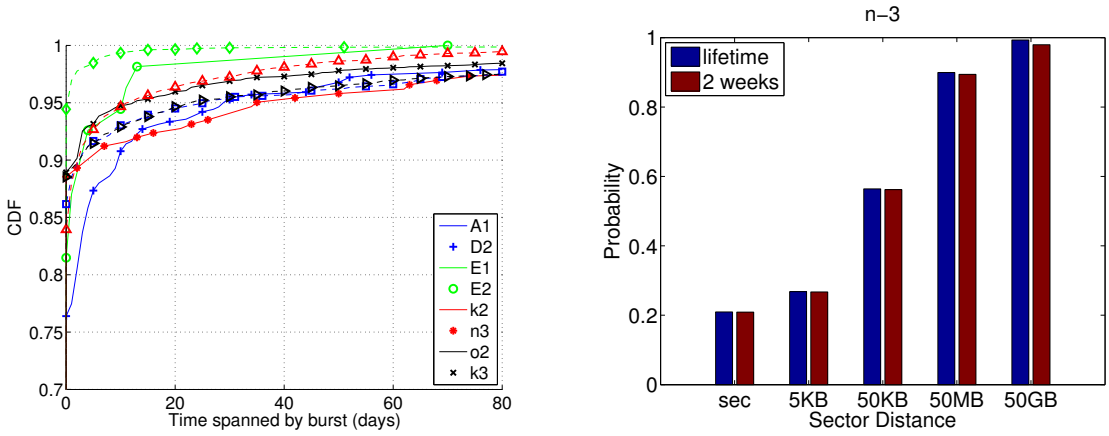


Figure 3: *Distribution of the time spanned by an error burst (left), and comparison of the probability of seeing another error within radius x in the 2 weeks after first error versus entire disk life (right)*

the use of intra-disk redundancy a system could achieve essentially the same reliability as that of a system operating without LSEs. Highly effective intra-disk redundancy might obviate the need for a background scrubber (and its potential impact on foreground traffic); in the best case, they might also enhance the reliability of a single parity RAID system sufficiently to make the use of double parity (e.g. RAID-4 or RAID-5) unnecessary, thereby avoiding the overheads and additional power usage of the second parity disk.

The intra-disk redundancy schemes we consider divide a disk into segments of k contiguous data sectors followed by m redundant sectors. The m redundant sectors are typically obtained using XOR-based operations on the data sectors. Different schemes vary in their reliability guarantees and their overhead depending on how the parity sectors are computed.

In our work, we evaluate 5 different intra-disk redundancy schemes. Three of the schemes (SPC, MDS, IPC) have been previously proposed, but have never been evaluated on field data. Two of the schemes are new schemes (MDS+SCP, CDP) that we suggest based on results from Section 3. All schemes are described below. We would like to note at this point, that while we do discuss the difference in overheads introduced by the different schemes, the focus of this section is to compare the relative degree of protection they can offer, rather than a detailed evaluation of their impact on performance.

Single parity check (SPC): A $k+1$ SPC scheme stores for each set of k contiguous data sectors one parity sector (typically a simple XOR on all data sectors). We refer to the set of k contiguous data sectors and the corresponding parity sector as a *parity group*. SPC schemes can tolerate a single error per parity group. Recovery from multiple errors in a parity group is only possible if there's an additional level of redundancy outside the disk (e.g. RAID). SPC schemes are simple and have little I/O overhead, since a write to a data sector requires only one additional write (to update the corresponding parity sector). However, a common concern is that due to spatial locality among sector errors, an error event will frequently affect multiple sectors in the same parity group.

Maximum distance separable (MDS) erasure codes: A $k+m$ MDS code consisting of k data sectors and m parity sectors can tolerate the loss of any m sectors in the segment. A well-known member of this code family are Reed-Solomon codes. While MDS codes are stronger than SPC they also create higher computational overheads (for example in the case of Reed-Solomon codes involving computations on Galois fields) and higher I/O overheads (for each write to a data sector all m parity sectors need to be updated). In most environments, these overheads make MDS codes impractical for use in intra-disk redundancy. Nevertheless, MDS codes provide an interesting upper bound on what reliability levels one can hope to achieve with intra-disk redundancy.

Interleaved parity check codes (IPC): A scheme proposed by Dholakia et al. [5, 10], specifically for use in intra-disk redundancy with lower overheads than MDS, but potentially weaker protection. The key idea is to ensure that the sectors within a parity group are spaced further apart than the length m of a typical burst of errors. A $k+m$ IPC achieves this by dividing k consecutive data sectors into $l = k/m$ segments of size m each, and imagining the $l \times m$ sectors $s_1, \ldots, s_{l \times m}$ layed out row-wise in an $l \times m$ matrix. Each one of the m parity sectors is computed as an XOR over one of the columns of this imaginary matrix, i.e. parity sector p_i is an XOR of $s_i, s_{i+m}, s_{i+2m}, \ldots, s_{i+(l-1)m}$. We refer to the data sectors in a column and the corresponding parity sector as a *parity*

Data Disk	Data Disk	Data Disk	Data Disk	Row Par. Disk	Diag. Par. Disk
0 (s_0)	1 (s_4)	2 (s_8)	3 (s_{12})	4 (p_0)	0 (p_4)
1 (s_1)	2 (s_5)	3 (s_9)	4 (s_{13})	0 (p_1)	1 (p_5)
2 (s_2)	3 (s_6)	4 (s_{10})	0 (s_{14})	1 (p_2)	2 (p_6)
3 (s_3)	4 (s_7)	0 (s_{11})	1 (s_{15})	2 (p_3)	3 (p_7)

Table 2: *Illustration of how to adapt RAID R-DP [4] with $p = 5$ for use in our intra-disk redundancy scheme CDP. The number in each block denotes the diagonal parity group a block belongs to. The parentheses show how an intra-disk redundancy segment with data sectors $s_0, ..., s_{15}$ and parity sectors $p_0, ..., p_7$ is mapped to the blocks in R-DP.*

group, and the $l \times m$ data sectors and the m parity sectors together as a *parity segment*. Observe, that all sectors in the same parity group have a distance of at least m. IPC can tolerate up to m errors provided they all affect different columns (and therefore different parity groups), but IPC can tolerate only a single error per column.

Hybrid SPC and MDS code (MDS+SPC): This scheme is motivated by Section 3.3, where we observed that for many models a disproportionately large fraction of all errors is concentrated in the first 5-15% of the logical block space. This scheme therefore uses a stronger (MDS) code for this first part of the drive, and a simple 8+1 SPC for the remainder of the drive.

Column Diagonal Parity (CDP): The motivation here is to provide a code that can tolerate a more diverse set of error patterns than IPC, but with less overhead than MDS. Our idea is to adapt the row-diagonal parity algorithm (R-DP) [4], which was developed to tolerate double disk failures in RAID, for use in intra-disk redundancy. R-DP uses $p+1$ disks, where p is a prime number, and assigns each data block to one row parity set and one diagonal parity set. R-DP uses $p-1$ disks for data, and two disks for row and diagonal parity. Figure 2 illustrates R-DP for $p = 5$. The row disk holds the parity for each row, and the number in each block denotes the diagonal parity group that the block belongs to.

We translate an R-DP scheme with parameter p to an intra-disk redundancy scheme with $k = (p-1)^2$ data sectors and $m = 2(p-1)$ parity sectors by mapping sectors to blocks as follows. We imagine traversing the matrix in Figure 2 column-wise and assigning the data sectors $s_0, ..., s_{15}$ consecutively to the blocks in the data disks and the parity sectors $p_0, ..., p_7$ to the blocks in the parity disks. The resulting assignment of sectors to blocks is shown in parentheses in the figure. Observe that without the diagonal parity, this scheme is identical to IPC: the row-parity of R-DP corresponds to to the parity sectors that IPC computes over the columns of the $(p-1) \times (p-1)$ matrix formed by rows of the data sectors. We therefore refer to our scheme as the column-diagonal parity (CDP) scheme.

CDP can tolerate any two error bursts of length $p-1$ that remove two full columns in Figure 2 (corresponding to two total disk failures in the R-DP scheme). In addition, CDP can tolerate a large number of other error patterns. Any data sector, whose corresponding column parity group has less than two errors or whose diagonal parity group has less than two errors, can be recovered[1]. Moreover, in many cases it will be possible to recover sectors where both the column parity group and the diagonal parity group have multiple errors, e.g. if the other errors in the column parity group can be recovered using their respective diagonal parity.

Note that for all codes there is a trade-off between the storage efficiency (i.e. $k/(k+m)$), the I/O overheads and the degree of protection a code can offer, depending on its parameter settings. Codes with higher storage efficiency generally have lower reliability guarantees. For a fixed storage efficiency, codes with larger parity segments provide stronger reliability for correlated errors that appear in bursts. At the same time, larger parity segments usually imply higher I/O overheads, since data sectors and the corresponding parity sectors are spaced further apart, requiring more disk head movement for updating parity sectors. The different schemes also differ in the flexibility that their parameters offer in controlling those trade-offs. For example, CDP cannot achieve any arbitrary combination of storage efficiency and parity segment size, since its only parameter p controls both the storage efficiency and the segment size.

4.1 Evaluation of redundancy schemes

4.1.1 Simple parity check (SPC) schemes

The question we want to answer in this section is what degree of protection simple parity check schemes can provide. Towards this end we simulate SPC schemes with varying storage efficiency, ranging from 1+1 to 128+1 schemes. While we explore the whole range of k from 1 to 128, in most applications the low storage efficiency of codes with values of k below 8 or 9 would probably render them impractical. Figure 4 shows the fraction of disks with uncorrectable errors (i.e. disks that have at least one parity group with multiple errors), the fraction of parity groups that have multiple errors, and the number of sectors per disk that cannot be recovered with SPC redundancy.

We observe that for values of k in the practically feasible range, a significant fraction of drives (about a quar-

[1] Exceptions are sectors in the diagonal parity group $p-1$, as R-DP stores no parity for this group.

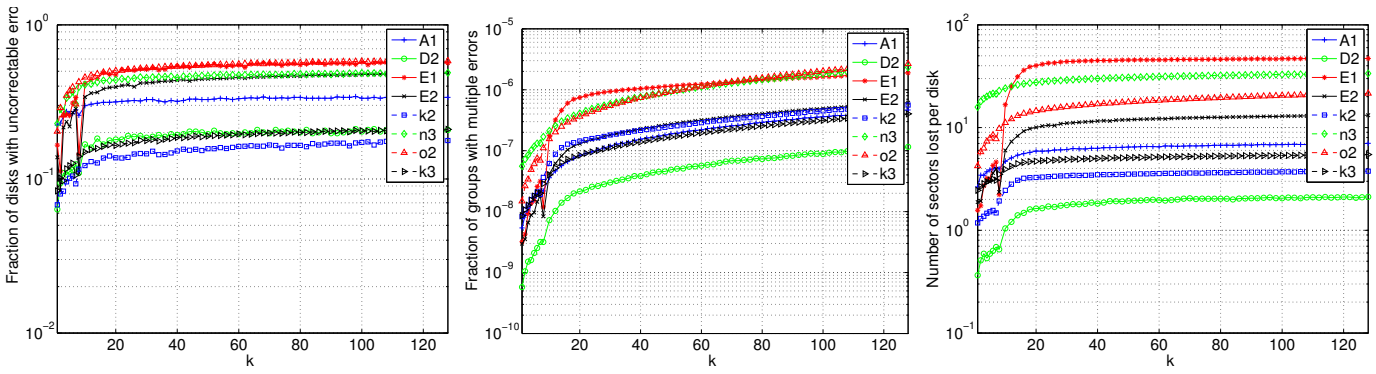


Figure 4: *Evaluation of $k+1$ SPC for different values of k. Fig. 4 (left) shows the fraction of disks with at least one uncorrectable error, i.e. disks that have at least one parity group with multiple errors; Fig. 4 (middle) shows the fraction of parity groups with multiple (and hence uncorrectable) errors; and Fig. 4 (right) shows the average number of sectors with uncorrectable errors per disk (due to multiple errors per parity group)*

ter averaged across all models) sees at least one uncorrectable error (i.e. a parity group with multiple errors). For some models (E-1, E-2, n-3, o-2) nearly 50% of drives see at least one uncorrectable error. On average more than 5 sectors per drive cannot be recovered with intra-disk redundancy. Even under the $1+1$ scheme, which sacrifices 50% of disk space for redundancy, on average 15% of disks have at least one parity group with multiple errors. It is noteworthy that there seems to be little difference in the results between enterprise and nearline drives.

The potential impact of multiple errors in a parity group depends on how close in time these errors occur. If there is ample time between the first and the second error in a group there is a high chance that either a background scrubber or an application access will expose and recover the first error, before the second error occurs. Figure 5 (left) shows the cumulative distribution function of the detection time between the first and the second error in parity groups with multiple errors. We observe that the time between the first two errors is small. More than 90% of errors are discovered within the same scrub interval (2 weeks, i.e. around 2.4×10^6 seconds). We conclude from Figure 5 that multiple errors in a parity group tend to occur at the same time, likely because they have been caused by the same event.

We are also interested in the distribution of the number of errors in groups that have multiple errors. If in most cases most of the sectors in a parity group are erroneous, even stronger protection schemes would not be able to recover those errors. On the other hand, if typically only a small number of sectors (e.g. 2 sectors) are bad, a slightly stronger code would be sufficient to recover those errors. Figure 5 (right) shows a histogram of the number of errors in parity groups with multiple errors for the 8+1 SPC scheme. We observe that across all models the most common case is that of double errors with about 50% of groups having two errors.

The above observations motivate us to look at stronger schemes in the next section.

4.1.2 More complex schemes

This section provides a comparative evaluation of IPC, MDS, CDP and SPC+MDS for varying segment sizes and varying degrees of storage efficiency. Larger segments have the potential for stronger data protection, as they space data and corresponding parity sectors further apart. At the same time larger segments lead to higher I/O overhead, as a write to a data sector requires updating the corresponding parity sector(s), which will require more head movement if the two are spaced further apart.

For CDP, the segment size and the storage efficiency are both determined by its parameter p (which has to be a prime number), while the other schemes are more flexible. In our first experiment we therefore start by varying p and adjusting the parameters of the other schemes to achieve the same m and k (i.e. $k = (p-1)^2$ and $m = 2(p-1)$). The bottom row in Figure 6 shows the results for p ranging from 5 to 23, corresponding to a range of storage efficiency from 66% to 92%, and segment sizes ranging from 24 to 528 sectors. In our second experiment, we keep the storage efficiency constant at 87% (i.e. on average 1 parity segment for 8 data segments), and explore different segment sizes by increasing m and k. The results are shown in the top row of Figure 6. For both experiments we show three different metrics: the fraction of disks with uncorrectable errors (graphs in left column), the average number of uncorrectable sectors per drive (middle column), and the fraction of parity

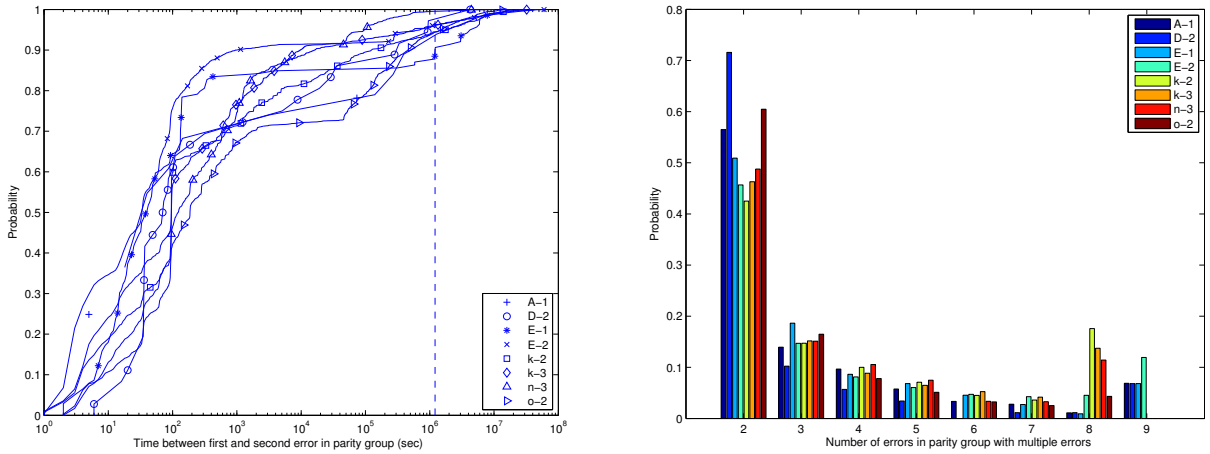


Figure 5: *Distribution of time between the first and second error in 8+1 SPC parity groups with multiple errors (left) and number of errors within a parity group with multiple errors for the case of an 8+1 SPC (right).*

segments with uncorrectable errors (right column).

We observe that all schemes provide clearly superior performance to SPC (for $m = 1$, IPC and MDS reduce to SPC). We also observe that MDS consistently provides the best performance, which might not be surprising as it is the scheme with the highest computational and I/O overheads. Among the remaining schemes CDP performs best, with improvements of an order of magnitude over IPC and SPC+MDS for larger p. SPC+MDS is not as strong, however its improvements of around 25% over simple SPC are impressive given that it applies stronger protection than SPC to only 10% of the total drive.

A surprising result might be the weak performance of IPC compared to MDS or CDP. The original papers [5, 10] proposing the idea of IPC predict the probability of data loss under IPC to be nearly identical to that of MDS. In contrast, we find that MDS (and CDP) consistently outperform IPC. For example, simply moving from an 8+1 to a 16+2 MDS scheme reduces nearly all metrics by 50%. Achieving similar results with an IPC scheme requires at least a 56+7 or 64+8 scheme. For larger segment sizes, MDS and CDP outperform IPC by an order of magnitude.

One might ask why IPC does not perform better. Based on our results in Section 3 we believe there are two reasons. First, the work in [5, 10] assumes that the only correlation between errors is that within an error burst and that different bursts are identically and independently distributed. However, as we saw in Section 3 there are significant correlations between errors that go beyond the correlation within a burst. Second, [5, 10] assumes that the length of error bursts follows a geometric distribution. Instead we found that the distribution of the length of error bursts has long tails (recall Figure 1) and is not fit well by a geometric distribution. As the authors observe in [10] the IPC scheme is sensitive to long tails in the distribution. The above observations underline the importance of using real-world data for modeling errors.

5 Proactive error detection with scrubbing

Scrubbing has been proposed as a mechanism for enhancing data reliability by proactively detecting errors [2, 12, 17]. Several commercial systems, including NetApp's, are making use of a background scrubber. A scrubber periodically reads the entire disk sequentially from the beginning to the end and uses inter-disk redundancy (e.g. provided by RAID) to correct errors. The scrubber runs continuously at a slow rate in the background as to limit the impact on foreground traffic, i.e. for a scrubbing interval s and drive capacity c, a drive is being scrubbed at a rate of c/s. Common scrub intervals are one or two weeks. We refer to a scrubber that works as described above as a *standard periodic scrubber*. In addition to standard periodic scrubbing, we investigate four additional policies.

Localized scrubbing: Given the spatial and temporal locality of LSEs, one idea for improving on standard periodic scrubbing is to take the detection of an error during a scrub interval as an indication that there are likely more errors in the neighborhood of this error. A scrubber could therefore decide upon the detection of an error to immediately scrub also the r sectors that follow the erroneous sector. These neighboring sectors are read at an accelerated rate a, rather than the default rate of c/s.

Accelerated scrubbing: This policy can be viewed as an extreme form of localized scrubbing: Once a bad sector is detected in a scrubbing interval, the entire remain-

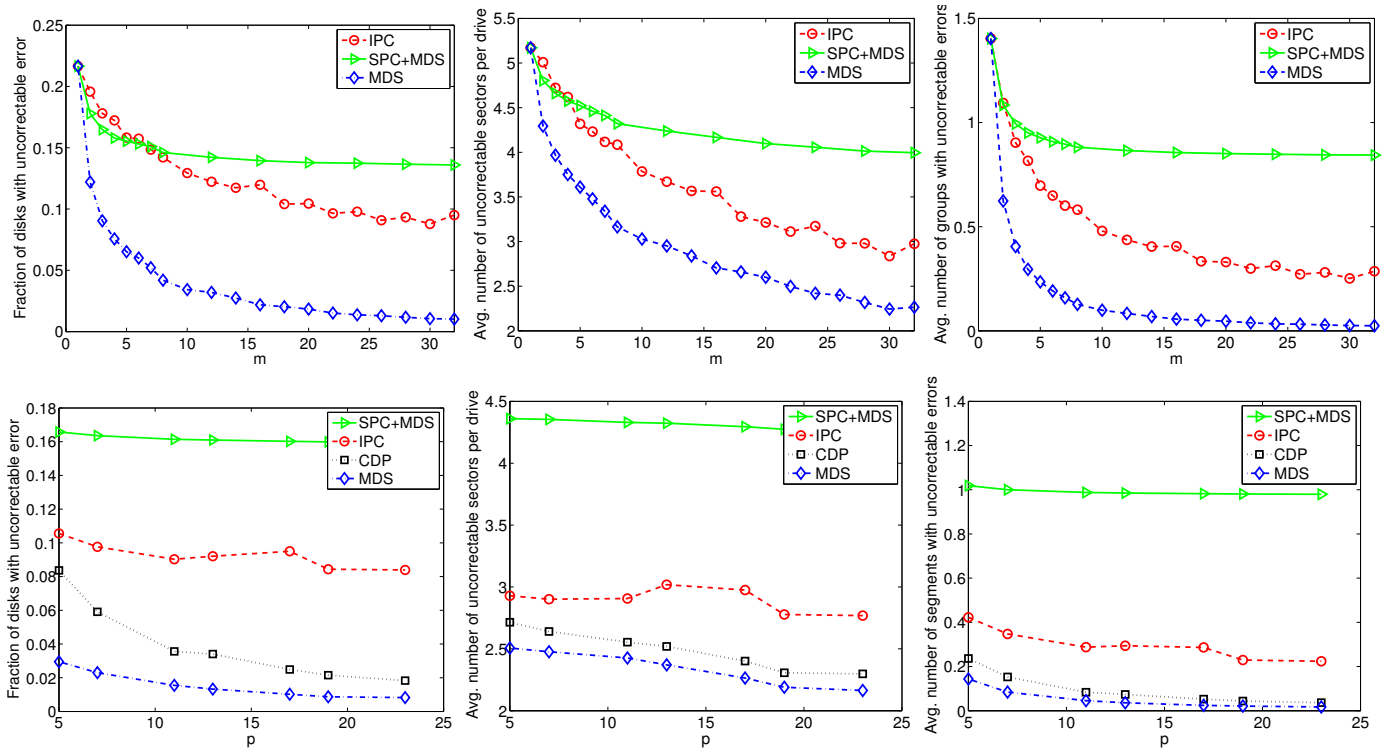


Figure 6: *Comparison of IPC, MDS, SPC+MDS, and CDP under three different metrics: the fraction of disks with at least one uncorrectable error (left), the number of sectors with unrecoverable errors per disk (middle), and the fraction of parity segments that have an unrecoverable error (right). In the top row, we keep the storage efficiency constant by varying m and adjusting $k = 8 \times m$. In the bottom row, we vary the p parameter of CDP and adjust all other policies to have the same m and k values, i.e. $k = (p-1)^2$ and $m = 2(p-1)$.*

der of the drive is scrubbed immediately at an accelerated rate a (rather than the default rate of c/s).

Staggered scrubbing: This policy has been proposed very recently by Oprea et al. [13] and aims to exploit the fact that errors happen in bursts. Rather than sequentially reading the disk from the beginning to the end, the idea is to quickly "probe" different regions of the drive, hoping that if a region of the drive has a burst of errors we will find one in the probe and immediately scrub the entire region. More formally, the drive is divided into r regions each of which is divided into segments of size s. In each scrub interval, the scrubber begins by reading the first segment of each region, then the second segment of each region, and so on. The policy uses the standard scrub rate of c/s and depends on two additional parameters, the segment size s and the number of regions r.

Accelerated staggered scrubbing: A combination of the two previous policies. We scrub segments in the order given by staggered scrubbing. Once we encounter an error in a region we immediately scrub the entire region at an increased scrub rate a (instead of the default c/s).

5.1 Evaluation methodology

Our goal is to evaluate the relative performance of the four different scrubbing policies described above. Any evaluation of scrubbing policies presents two difficulties.

First, the performance of a scrub policy will critically depend on the temporal and spatial properties of errors. While our data contain logical sector numbers and timestamps for each reported LSE, the timestamps correspond to the time when an error was detected, not necessarily the time when it actually happened. While we have no way of knowing the exact time when an error happened we will use three different methods for approximating this time. All methods rely on the fact that we know the time window during which an error must have happened: since the scrub interval on NetApp's systems is two weeks, an error can be latent for at most 2 weeks before it is detected. Hence an error must have happened within 2 weeks before the timestamp in the trace. In addition to running simulations directly on the trace we use the three methods below for approximating timestamps:

Method 1: The strong spatial and temporal locality observed in Section 3 indicate that errors that are

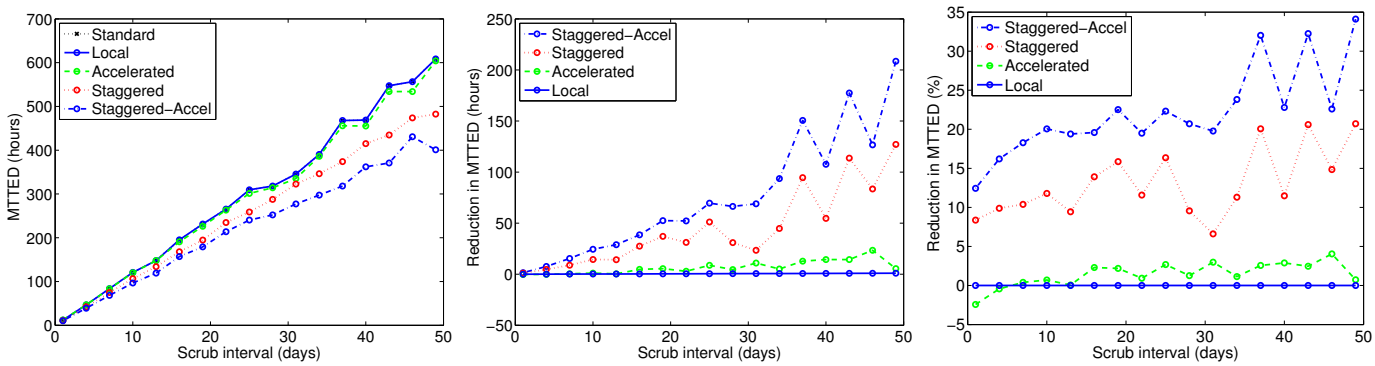


Figure 7: *Comparison of all policies for varying scrub intervals (results averaged across all disk models)*

detected within the same scrub period are likely to be caused by the same error event (e.g. a scratch in the surface or a high-fly write). Method 1 assumes that all errors that happened within a radius of 50MB of each other in the same scrub interval were caused by the same event and assigns all these errors the same timestamp (the timestamp of the error that was detected first).

Method 2: This method goes one step further and assumes that *all* errors that are reported in the same scrub interval happened at the same time (not an unlikely assumption, recall Figure 3) and assigns all of them the timestamp of the first error in the scrub interval.

Method 3: The last method takes an adversary's stance and makes the (unlikely) assumption that all errors in a scrub interval happened completely independently of each other and assigns each error a timestamp that lies randomly in the 2-week interval before the error was detected.

The second difficulty in evaluating scrubbing policies is that there is a possibility that the scrubbing frequency itself affects the rate at which errors happen, i.e. the additional workload created by frequent scrubbing might cause additional errors. After talking to vendors and studying reports [6, 7] on the common error modes leading to LSEs, it seems unlikely that the read frequency in a system (in contrast to the write frequency) would have a major impact on errors. The majority of reported error modes are either directly related to writes (such as high-fly writes) or can happen whenever the disk is spinning, independent of whether data is being read or written (such as thermal asperities, corrosion, and scratches or smears). Nevertheless we are hesitant to assume that the scrub frequency has zero impact on the error rate. Since the goal of our study is not to determine the optimal scrub frequency, but rather to evaluate the relative performance of the different policies, we only compare the performance of different policies under the same scrub frequency. This way, all policies would be equally affected by an increase in errors caused by additional reads.

The main metric we use to evaluate the effectiveness of a scrub policy is the mean time to error detection (MTTED). The MTTED will be a function of the scrub interval since for all policies more frequent scrubs are expected to lead to shorter detection times.

5.2 Comparison of scrub policies

Figure 7 shows a comparison of the four different scrub policies described in the beginning of this section. The graphs, from left to right, show the mean time to error detection (MTTED), the reduction in MTTED (in hours) that each policy provides over standard periodic scrubbing, and the percentage improvement in MTTED over standard periodic scrubbing. We vary the scrub interval from one day to 50 days. The scrub radius in the local policy is set to 128MB. The accelerated scrub rate a for all policies is set to 7000 sectors/sec, which is two times slower than the read performance[2] reported for scrubs in [13]. For the staggered policies we chose a region size of 128MB and a segment size of 1MB (as suggested in [13]). We later also experiment with other parameter choices for the local and the staggered scrub algorithms. When generating the graphs in Figure 7, we took the timestamps verbatim from the trace. In Section 5.2.4 we will discuss how the results change when we use one of the three methods for approximating timestamps, as described in Section 5.1.

5.2.1 Local scrubbing

The performance of the local scrub policy turns out to be disappointing, being virtually identical to that of standard scrubbing. We explain this with the fact that its only potential for improvements lies in getting faster to errors that are within a 128MB radius of a previously detected

[2]The SCSI verify command used in scrubs is faster than a read operation as no data is transferred, so this estimate should be conservative.

error. However, errors within this close neighborhood will also be detected quickly by the standard sequential scrubber (as they are in the immediate neighborhood).

To evaluate the broader potential of local scrubbing, we experimented with different radii, to see whether this yields larger improvements. We find that only for very large radii (on the order of several GB) the results are significant and even then only some of the models show improvements of more than 10%.

5.2.2 Accelerated scrubbing

Similar to local scrubbing, also accelerated scrubbing (without staggering) does not yield substantial improvements. The reasons are likely the same as those for local scrubbing. Once it encounters an error, accelerated scrubbing will find subsequent errors quicker. However, due to spatial locality most of the subsequent errors will be in the close neighborhood of the first and will also be detected soon by standard scrubbing. We conclude that the main weakness of local and accelerated scrubbing is that they only try to minimize the time to find additional errors, once the first error has been found. On the other hand, staggered scrubbing minimizes the time it takes to determine whether there are any errors and in which part of the drive they are.

5.2.3 Staggered scrubbing

We observe that the two staggered policies both provide significant improvements over standard scrubbing for all scrubbing frequencies. For commonly used intervals in the 7-14 day range, improvements in MTTED for these policies range from 30 to 70 hours, corresponding to an improvement of 10–20%. These improvements increase with larger scrubbing intervals. We also note that even simple (non-accelerated) staggered scrubbing yields significantly better performance than both local or accelerated scrubbing, without using any accelerated I/Os.

Encouraged by the good performance of staggered scrubbing, we take a closer look at the impact of the choice of parameters on its effectiveness, in particular the choice of the segment size, as this parameter can greatly affect the overheads associated with staggered scrubbing. From the point of view of minimizing overhead introduced by the scrubber, one would like to choose the segments as large as possible, since the sectors in individual segments are read through fast sequential I/Os, while moving between a large number of small segments requires slow random I/Os. On the other hand if the size of segments becomes extremely large, the effectiveness of staggered scrubbing in detecting errors early will approach that of standard scrubbing (the extreme case of one segment per region leads to a policy identical to standard scrubbing.)

We explore the effect of the segment size for several different region sizes. Interestingly, we find consistently for all region sizes that the segment size has a relatively small effect on performance. As a rough rule of thumb, we observe that scrubbing effectiveness is not negatively affected as long as the segment size is smaller than a quarter to one half of the size of a region. For example, for a region size of 128MB, we find the effectiveness of scrubbing to be identical for segment sizes ranging from 1KB to 32MB. For a segment size of 64MB, the level of improvement that staggered scrubbing offers over standard scrubbing drops by 50%. Oprea [13] reports experimental results showing that for segment sizes of 1MB and up, the I/O overheads of staggered scrubbing are comparable to that of standard scrubbing. That means there is a large range of segment sizes that are practically feasible and also effective in reducing MTTED.

5.2.4 Approximating timestamps

In our simulation results in Figure 7, we assume that the timestamps in our traces denote the actual times when errors happened, rather than the time when they were detected. We also repeated all experiments with the three methods for approximating timestamps described in Section 5.1.

We find that under the two methods that try to make realistic assumptions about the time when errors happened, based on the spatio-temporal correlations we observed in Section 3, the performance improvements of the scrub policies compared to standard scrubbing either stays the same or increases. When following method 1 (all errors detected in the same scrub interval within a 50MB-radius are assigned the same timestamp), the improvements of staggered accelerated scrubbing increase significantly, for some models as much as 50%, while the performance of all other policies stays the same. When following method 2 (all errors within the same scrub interval are assigned the same timestamp) all methods see a slight increase of around 5% in their gains compared to standard scrubbing. When making the (unrealistic) worst case assumption of method 3 that errors are completely uncorrelated in time, the performance improvements of all policies compared to standard scrubbing drop significantly. Local and accelerated scrubbing show no improvements, and the MTTDE reduction of staggered scrubbing and accelerated staggered scrubbing drops to 2–5%.

6 Summary and discussion

The main contributions of this paper are a detailed statistical analysis of field data on latent sector errors and a comparative evaluation of different approaches for protecting against LSEs, including some new schemes that we propose based on our data analysis.

The statistical analysis revealed some interesting properties. We observe that many of the statistical aspects of LSEs are well modeled by power-laws, including the length of error bursts (i.e. a series of contiguous sectors affected by LSEs), the number of good sectors that separate error bursts, and the number of LSEs observed per time. We find that these properties are poorly modeled by the most commonly used distributions, geometric and Poisson. Instead we observe that a Pareto distribution fits the data very well and report the parameters that provide the best fit. We hope this data will be useful for other researchers who do not have access to field data. We find no significant difference in the statistical properties of LSEs in nearline drives versus enterprise class drives.

Some of our statistical observations might also hold some clues as to what mechanisms cause LSEs. For example, we observe that nearly all drives with LSEs, experience all LSEs in their lifetime within the same 2-week period, indicating that for most drives most errors have been caused by the same event (e.g. one scratch), rather than a slow and continuous wear-out of the media.

An immediate implication of the above observation is that both approaches commonly used to model LSEs are unrealistic. The first approach ties LSE arrivals to the workload process, by assuming a certain bit error rate, and assuming that each read or write has the same fixed probability p of causing an LSE. The second approach models LSEs by a separate arrival process, most commonly a Poisson process. Both will result in a much smoother process than the one seen in practice.

In our comparative study of the effectiveness of intra-disk redundancy schemes we find that simple parity check (SPC) schemes still leave a significant fraction of drives (50% for some models) with errors that cannot be recovered by intra-disk redundancy. An observation in our statistical study that a large fraction of errors (for some models 40%) is concentrated in a small area of the drive (the bottom 10% of the logical sector space) leads us to a new scheme that uses stronger codes for only this part of the drive and reduces the number of drives with unrecoverable errors by 30% compared to SPC.

We also evaluate the interleaved-parity check (IPC) scheme [5, 10] that promises reliability close to the powerful maximum distance separable erasure codes (MDS), with much less overhead. Unfortunately, we find IPC's reliability to be significantly weaker than that of MDS. We attribute the discrepancy between our results and those in [5, 10] to the difference between the statistical assumptions (e.g. geometric distribution of error bursts) in [5,10] and the properties of LSEs in the field (long tails in error burst distributions). Finally, we present a new scheme, based on adaptations of the ideas behind row-diagonal parity [4], with significantly lower overheads than MDS, but very similar reliability.

In our analysis of scrubbing policies, we find that a simple policy, staggered scrubbing [13], can improve the mean time to error detection by up to 40%, compared to standard sequential scrubbing. Staggered scrubbing achieves these results just by changing the order in which sectors are scrubbed, without changing the scrub frequency or introducing significant I/O overhead.

Our work opens up a number of avenues for future work. Our long-term goal is to understand how scrubbing and intra-disk redundancy interact with the redundancy provided by RAID, how different redundancy layers should be integrated, and to quantify how different approaches affect the actual mean time to data loss. Answering these questions is not easy, as it will require a complete statistical model that captures spatial and temporal locality, and total disk failures, as well as LSEs.

7 Acknowledgments

We would like to thank the Advanced Development Group at Network Appliances for collecting and sharing the data used in this study. In particular, we thank Lakshmi Bairavasundaram, Garth Goodson, Shankar Pasupathy and Jiri Schindler for answering questions about the data and their systems and for providing feedback on the first drafts of the paper. The first author would like to thank the System Health Group at Google for hosting her during the summer of 2009 and for providing the opportunity to analyze the SMART data collected on their hard disk drives. We also thank Jay Wylie for sharing his insights regarding intra-disk redundancy codes and the anonymous reviewers for their useful feedback. This work was supported in part by an NSERC Discovery grant.

References

[1] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *Proc. of SIGMETRICS'07*, 2007.

[2] M. Baker, M. Shah, D. S. H. Rosenthal, M. Roussopoulos, P. Maniatis, T. Giuli, and P. Bungale. A fresh look at the reliability of long-term digital storage. In *Proc. of EuroSys '06*, 2006.

[3] M. Blaum, J. Brady, J. Bruck, and J. Menon. Evenodd: an optimal scheme for tolerating double disk failures in RAID architectures. In *Proc. of ISCA '94*, 1994.

[4] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-diagonal parity for double disk failure correction. In *Proc. of FAST '04*, 2004.

[5] A. Dholakia, E. Eleftheriou, X.-Y. Hu, I. Iliadis, J. Menon, and K. K. Rao. A new intra-disk redundancy scheme for high-reliability RAID storage systems in the presence of unrecoverable errors. *ACM TOS*, 4(1), 2008.

[6] J. G. Elerath. Hard-disk drives: the good, the bad, and the ugly. *Commun. ACM*, 52(6), 2009.

[7] J. G. Elerath and M. Pecht. Enhanced reliability modeling of RAID storage systems. In *Proc. of DSN '07*, 2007.

[8] J. L. Hafner. Weaver codes: highly fault tolerant erasure codes for storage systems. In *Proc. of FAST'05*, 2005.

[9] J. L. Hafner. Hover erasure codes for disk arrays. In *Proc. of DSN '06*, 2006.

[10] I. Iliadis, R. Haas, X.-Y. Hu, and E. Eleftheriou. Disk scrubbing versus intra-disk redundancy for high-reliability raid storage systems. In *SIGMETRICS'08*, 2008.

[11] M. K. Mckusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *ACM TOCS*, 2(3), 1984.

[12] N. Mi, A. Riska, E. Smirni, and E. Riedel. Enhancing data availability in disk drives through background activities. In *Proc. of DSN*, 2008.

[13] A. Oprea and A. Juels. A clean-slate look at disk scrubbing. In *Proc. of FAST '10*, 2010.

[14] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proc. of SIGMOD*, 1988.

[15] J. S. Plank. The RAID-6 Liberation codes. In *Proc. of FAST'08*, 2008.

[16] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON File Systems. In *Proc. of SOSP '05*, 2005.

[17] T. Schwarz, Q. Xin, E. L. Miller, D. E. Long, A. Hospodor, and S. Ng. Disk scrubbing in large archival storage systems. In *Proc. of MASCOTS '04*, 2004.

[18] J. J. Wylie and R. Swaminathan. Determining fault tolerance of XOR-based erasure codes efficiently. In *Proc. of DSN'07*, 2007.

DFS: A File System for Virtualized Flash Storage

William K. Josephson wkj@CS.Princeton.EDU
Lars A. Bongo larsab@Princeton.EDU
David Flynn dflynn@FusionIO.COM
Kai Li li@CS.Princeton.EDU

Abstract

This paper presents the design, implementation and evaluation of Direct File System (DFS) for virtualized flash storage. Instead of using traditional layers of abstraction, our layers of abstraction are designed for directly accessing flash memory devices. DFS has two main novel features. First, it lays out its files directly in a very large virtual storage address space provided by FusionIO's virtual flash storage layer. Second, it leverages the virtual flash storage layer to perform block allocations and atomic updates. As a result, DFS performs better and it is much simpler than a traditional Unix file system with similar functionalities. Our microbenchmark results show that DFS can deliver 94,000 I/O operations per second (IOPS) for direct reads and 71,000 IOPS for direct writes with the virtualized flash storage layer on FusionIO's ioDrive. For direct access performance, DFS is consistently better than ext3 on the same platform, sometimes by 20%. For buffered access performance, DFS is also consistently better than ext3, and sometimes by over 149%. Our application benchmarks show that DFS outperforms ext3 by 7% to 250% while requiring less CPU power.

1 Introduction

Flash memory has traditionally been the province of embedded and portable consumer devices. Recently, there has been significant interest in using it to run primary file systems for laptops as well as file servers in data centers. Compared with magnetic disk drives, flash can substantially improve reliability and random I/O performance while reducing power consumption. However, these file systems are originally designed for magnetic disks which may not be optimal for flash memory. A key systems design question is to understand how to build the entire system stack including the file system for flash memory.

Past research work has focused on building firmware and software to support traditional layers of abstractions for backward compatibility. For example, recently proposed techniques such as the flash translation layer (FTL) are typically implemented in a solid state disk controller with the disk drive abstraction [5, 6, 26, 3]. Systems software then uses a traditional block storage interface to support file systems and database systems designed and optimized for magnetic disk drives. Since flash memory is substantially different from magnetic disks, the rationale of our work is to study how to design new abstraction layers including a file system to exploit the potential of NAND flash memory.

This paper presents the design, implementation, and evaluation of the Direct File System (DFS) and describes the virtualized flash memory abstraction layer it uses for FusionIO's ioDrive hardware. The virtualized storage abstraction layer provides a very large, virtualized block addressed space, which can greatly simplify the design of a file system while providing backward compatibility with the traditional block storage interface. Instead of pushing the flash translation layer into disk controllers, this layer combines virtualization with intelligent translation and allocation strategies for hiding bulk erasure latencies and performing wear leveling.

DFS is designed to take advantage of the virtualized flash storage layer for simplicity and performance. A traditional file system is known to be complex and typically requires four or more years to become mature. The complexity is largely due to three factors: complex storage block allocation strategies, sophisticated buffer cache designs, and methods to make the file system crash-recoverable. DFS dramatically simplifies all three aspects. It uses virtualized storage spaces *directly* as a true single-level store and leverages the virtual to physical block allocations in the virtualized flash storage layer to avoid explicit file block allocations and reclamations. By doing so, DFS uses extremely simple metadata and data layout. As a result, DFS has a short datapath to flash memory and encourages users to access data directly instead of going through a large and complex buffer cache. DFS leverages the atomic update feature of the virtualized flash storage layer to achieve crash recovery.

We have implemented DFS for the FusionIO's virtualized flash storage layer and evaluated it with a suite of benchmarks. We have shown that DFS has two main advantages over the ext3 filesystem. First, our file sys-

tem implementation is about one eighth that of ext3 with similar functionality. Second, DFS has much better performance than ext3 while using the same memory resources and less CPU. Our microbenchmark results show that DFS can deliver 94,000 I/O operations per second (IOPS) for direct reads and 71,000 IOPS direct writes with the virtualized flash storage layer on FusionIO's ioDrive. For direct access performance, DFS is consistently better than ext3 on the same platform, sometimes by 20%. For buffered access performance, DFS is also consistently better than ext3, and sometimes by over 149%. Our application benchmarks show that DFS outperforms ext3 by 7% to 250% while requiring less CPU power.

2 Background and Related Work

In order to present the details of our design, we first provide some background on flash memory and the challenges to using it in storage systems. We then provide an overview of related work.

2.1 NAND Flash Memory

Flash memory is a type of electrically erasable solid-state memory that has become the dominant technology for applications that require large amounts of non-volatile solid-state storage. These applications include music players, cell phones, digital cameras, and shock sensitive applications in the aerospace industry.

Flash memory consists of an array of individual cells, each of which is constructed from a single floating-gate transistor. Single Level Cell (SLC) flash stores a single bit per cell and is typically more robust; Multi-Level Cell (MLC) flash offers higher density and therefore lower cost per bit. Both forms support three operations: read, write (or program), and erase. In order to change the value stored in a flash cell it is necessary to perform an erase before writing new data. Read and write operations typically take tens of microseconds whereas the erase operation may take more than a millisecond.

The memory cells in a NAND flash device are arranged into pages which vary in size from 512 bytes to as much as 16KB each. Read and write operations are page-oriented. NAND flash pages are further organized into erase blocks, which range in size from tens of kilobytes to megabytes. Erase operations apply only to entire erase blocks; any data in an erase block that is to be preserved must be copied.

There are two main challenges in building storage systems using NAND flash. The first is that an erase operation typically takes about one or two milliseconds. The second is that an erase block may be erased successfully only a limited number of times. The endurance of an erase block depends upon a number of factors, but usually ranges from as little as 5,000 cycles for consumer grade MLC NAND flash to 100,000 or more cycles for enterprise grade SLC NAND flash.

2.2 Related Work

Douglis *et al.* studied the effects of using flash memory without a special software stack [11]. They showed that flash could improve read performance by an order of magnitude and decrease energy consumption by 90%, but that due to bulk erasure latency, write performance also decreased by a factor of ten. They further noted that large erasure block size causes unnecessary copies for cleaning, an effect often referred to as "write amplification".

Kawaguchi *et al.* [14] describe a transparent device driver that presents flash as a disk drive. The driver dynamically maps logical blocks to physical addresses, provides wear-leveling, and hides bulk erasure latencies using a log-structured approach similar to that of LFS [27]. State-of-the art implementations of this idea, typically called the Flash Translation Layer, have been implemented in the controllers of several high-performance Solid State Drives (SSDs) [3, 16].

More recent efforts focus on high-performance in SSDs, particularly for random writes. Birrell *et al.* [6], for instance, describe a design that significantly improves random write performance by keeping a fine-grained mapping between logical blocks and physical flash addresses in RAM. Similarly, Agrawal *et al.* [5] argue that SSD performance and longevity is strongly workload dependent and further that many systems problems that previously have appeared higher in the storage stack are now relevant to the device and its firmware. This observation has lead to the investigation of buffer management policies for a variety of workloads. Some policies, such as Clean First LRU (CFLRU) [24] trade off a reduced number of writes for additional reads. Others, such as Block Padding Least Recently Used (BPLRU) [15] are designed to improve performance for fine-grained updates or random writes.

eNVy [33] is an early file system design effort for flash memory. It uses flash memory as fast storage, a battery-backed SRAM module as a non-volatile cache for combining writes into the same flash block for performance, and copy-on-write page management to deal with bulk erasures

More recently, a number of file systems have been designed specifically for flash memory devices. YAFFS, JFFS2, and LogFS [19, 32] are example efforts that hide bulk erasure latencies and perform wear-leveling of NAND flash memory devices at the file system level using the log-structured approach. These file systems were initially designed for embedded applications instead of high-performance applications and are not generally suitable for use with the current generation of high-performance

flash devices. For instance, YAFFS and JFFS2 manage raw NAND flash arrays directly. Furthermore, JFFS2 must scan the entire physical device at mount time which can take many minutes on large devices. All three filesystems are designed to access NAND flash chips directly, negating the performance advantages of the hardware and software in emerging flash device. LogFS does have some support for a block-device compatibility mode that can be used as a fall-back at the expense of performance, but none are designed to take advantage of emerging flash storage devices which perform their own flash management.

3 Our Approach

This section presents the three main aspects of our approach: (a) new layers of abstraction for flash memory storage systems which yield substantial benefits in simplicity and performance; (b) a virtualized flash storage layer, which provides a very large address space and implements dynamic mapping to hide bulk erasure latencies and to perform wear leveling; and (c) the design of DFS which takes full advantage of the virtualized flash storage layer. We further show that DFS is simple and performs better than the popular Linux ext3 file system.

3.1 Existing vs. New Abstraction Layers

Figure 1 shows the architecture block diagrams for existing flash storage systems and our proposed architecture. The traditional approach is to package flash memory as a solid-state disk (SSD) that exports a disk interface such as SATA or SCSI. An advanced SSD implements a flash translation layer (FTL) in its controller that maintains a dynamic mapping from logical blocks to physical flash pages to hide bulk erasure latencies and to perform wear leveling. Since a SSD uses the same interface as a magnetic disk drive, it supports the traditional block storage software layer which can be either a simple device driver or a sophisticated volume manager. The block storage layer then supports traditional file systems, database systems, and other software designed for magnetic disk drives. This approach has the advantage of disrupting neither the application-kernel interface nor the kernel-physical storage interface. On the other hand, it has a relatively thick software stack and makes it difficult for the software layers and hardware to take full advantage of the benefits of flash memory.

We advocate an architecture in which a greatly simplified file system is built on top of a virtualized flash storage layer implemented by the cooperation of the device driver and novel flash storage controller hardware. The controller exposes direct access to flash memory chips to the virtualized flash storage layer.

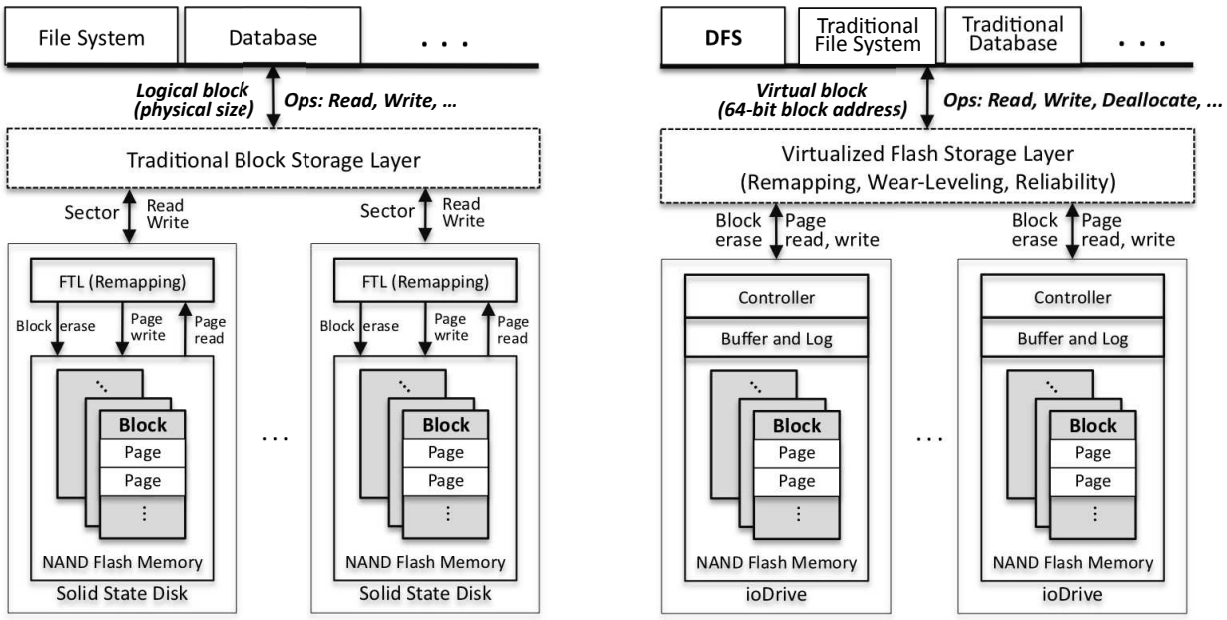
The virtualized flash storage layer is implemented at the device driver level which can freely cooperate with specific hardware support offered by the flash memory controller. The virtualized flash storage layer implements a large virtual block addressed space and maps it to physical flash pages. It handles multiple flash devices and uses a log-structured allocation strategy to hide bulk erasure latencies, perform wear leveling, and handle bad page recovery. This approach combines the virtualization and FTL together instead of pushing FTL into the disk controller layer. The virtualized flash storage layer can still provide backward compatibility to run existing file systems and database systems. The existing software can benefit from the intelligence in the device driver and hardware rather than having to implement that functionality independently in order to use flash memory. More importantly, flash devices are free to export a richer interface than that exposed by disk-based interfaces.

Direct File System (DFS) is designed to utilize the functionality provided by the virtualized flash storage layer. In addition to leveraging the support for wear-leveling and for hiding the latency of bulk erasures, DFS uses the virtualized flash storage layer to perform file block allocations and reclamations and uses atomic flash page updates for crash recovery. This architecture allows the virtualized flash storage layer to provide an object-based interface. Our main observation is that the separation of the file system from block allocations allows the storage hardware and block management algorithms to evolve jointly and independently from the file system and user-level applications. This approach makes it easier for the block management algorithms to take advantage of improvements in the underlying storage subsystem.

3.2 Virtualized Flash Storage Layer

The virtual flash storage layer provides an abstraction to enable client software such as file systems and database systems to take advantage of flash memory devices while providing backward compatibility with the traditional block storage interface. The primary novel feature of the virtualized flash storage layer is the provision for a very large, virtual block-addressed space. There are three reasons for this design. First, it provides client software with the flexibility to directly access flash memory in a single level store fashion across multiple flash memory devices. Second, it hides the details of the mapping from virtual to physical flash memory pages. Third, the flat virtual block-addressed space provides clients with a backward compatible block storage interface.

The mapping from virtual blocks to physical flash memory pages deals with several flash memory issues. Flash memory pages are dynamically allocated and reclaimed to hide the latency of bulk erasures, to distribute



(a) Traditional layers of abstractions **(b) Our layers of abstractions**

Figure 1: Flash Storage Abstractions

writes evenly to physical pages for wear-leveling, and to detect and recover bad pages to achieve high reliability. Unlike a conventional Flash Translation Layer (FTL), the mapping supports a very large number of virtual pages – orders-of-magnitude larger than the available physical flash memory pages.

The virtualized flash storage layer currently supports three operations: read, write, and trim or deallocate. All operations are block-based operations, and the block size in the current implementation is 512 bytes. The write operation triggers a dynamic mapping from a virtual to physical page, thus there is no explicit allocation operation. The deallocate operation deallocates a range of virtual addresses. It removes the mappings of all mapped physical flash pages in the range and hands them to a garbage collector to recycle for future use. We anticipate that future versions of the VFSL will also support a move operation to allow data to be moved from one virtual address to another without incurring the cost of a read, write, and deallocate operation for each block to be copied.

The current implementation of the virtualized flash storage layer is a combination of a Linux device driver and FusionIO's ioDrive special purpose hardware. The ioDrive is a PCI Express card densely populated with either 160GB or 320GB of SLC NAND flash memory. The software for the virtualized flash storage layer is implemented as a device driver in the host operating system and leverages hardware support from the ioDrive itself.

The ioDrive uses a novel partitioning of the virtualized flash storage layer between the hardware and device driver to achieve high performance. The overarching design philosophy is to separate the data and control paths and to implement the control path in the device driver and the data path in hardware. The data path on the ioDrive card contains numerous individual flash memory packages arranged in parallel and connected to the host via PCI Express. As a consequence, the device achieves highest throughput with moderate parallelism in the I/O request stream. The use of PCI Express rather than an existing storage interface such as SCSI or SATA simplifies the partitioning of control and data paths between the hardware and device driver.

The device provides hardware support of checksum generation and checking to allow for the detection and correction of errors in case of the failure of individual flash chips. Metadata is stored on the device in terms of physical addresses rather than virtual addresses in order to simplify the hardware and allow greater throughput at lower economic cost. While individual flash pages are relatively small (512 bytes), erase blocks are several megabytes in size in order to amortize the cost of bulk erase operations.

The mapping between virtual and physical addresses is maintained by the kernel device driver. The mapping between 64-bit virtual addresses and physical addresses is maintained using a variation on B-trees in memory. Each address points to a 512-byte flash memory page, allowing a virtual address space of 2^{73} bytes. Updates are made stable by recording them in a log-structured fashion: the hardware interface is append-only. The device driver is also responsible for reclaiming unused storage using a garbage collection algorithm. Bulk erasure scheduling and wear leveling algorithms for flash endurance are integrated into the garbage collection component of the device driver.

A primary rationale for implementing the virtual to physical address translation and garbage collection in the device driver rather than in an embedded processor on the ioDrive itself is that the device driver can automatically take advantage of improvements in processor and memory bus performance on commodity hardware without requiring significant design work on a proprietary embedded platform. This approach does have the drawback of requiring potentially significant processor and memory resources on the host.

3.3 DFS

DFS is a full-fledged implementation of a Unix file system and it is designed to take advantage of several features of the virtualized flash storage layer, including large virtualized address space, direct flash access and its crash recovery mechanism. The implementation runs as a loadable kernel module in the Linux 2.6 kernel. The DFS kernel module implements the traditional Unix file system APIs via the Linux VFS layer. It supports the usual methods such as open, close, read, write, pread, pwrite, lseek, and mmap. The Linux kernel requires basic memory mapped I/O support in order to facilitate the execution of binaries residing on DFS file systems.

3.3.1 Leveraging Virtualized Flash Storage

DFS delegates I-node and file data block allocations and deallocations to the virtualized flash storage layer. The virtualized flash storage layer is responsible for block allocations and deallocations, for hiding the latency of bulk erasures, and for wear leveling.

We have considered two design alternatives. The first is to let the virtualized storage layer export an object-based interface. In this case, a separate object is used to represent each file system object and the virtualized flash storage layer is responsible for managing the underlying flash blocks. The main advantage of this approach is that it can provide a close match with what a file system implementation needs. The main disadvantage is the complexity of an object-based interface that provides backwards compatibility with the traditional block storage interface.

The second is to ask the virtualized flash storage layer to implement a large logical address space that is sparse. Each file system object will be assigned a contiguous range of logical block addresses. The main advantages of this approach are its simplicity and its natural support for the backward compatibility with the traditional block storage interface. The drawback of this approach is its potential waste of the virtual address space. DFS has taken this approach for its simplicity.

We have configured the ioDrive to export a sparse 64-bit logical block address space. Since each block contains 512 bytes, the logical address space spans 2^{73} bytes. DFS can then use this logical address space to map file system objects to physical storage.

DFS allocates virtual address space in contiguous "allocation chunks". The size of these chunks is configurable at file system initialization time but is 2^{32} blocks or 2TB by default. User files and directories are partitioned into two types: large and small. A large file occupies an entire chunk whereas multiple small files reside in a single chunk. When a small file grows to become a large file, it is moved to a freshly allocated chunk. The current implementation must implement this by copying the file contents, but we anticipate that future versions of the virtual flash storage layer will support changing the virtual to physical translation map without having to copy data. The current implementation does not support remapping large files into the small file range should a file shrink.

When the filesystem is initialized, two parameters must be chosen: the maximum size of a small file, which must be a power of two, and the size of allocation chunks, which is also the maximum size of a large file. These two parameters are fixed once the filesystem is initialized. They can be chosen in a principled manner given the anticipated workload. There have been many studies of file size distributions in different environments, for instance those by Tannenbaum *et al.* [28] and Docuer and Bolosky [10]. By default, small files are those less than 32KB.

The current DFS implementation uses a 32-bit I-node number to identify individual files and directories and a 32-bit block offset into a file. This means that DFS can support up to $-1 + 2^{32}$ files and directories in total since the first I-node number is reserved for the system. The largest supported file size is 2TB with 512-byte blocks since the block offset is 32 bits. The I-node itself stores the base virtual address for the logical extent containing the file data. This base address together with the file offset identifies the virtual address of a file block. Figure 2 depicts the mapping from file descriptor and offset to logical block address in DFS.

The very simple mapping from file and offset to logical block address has another beneficial implication. Each file is represented by a single logical extent, making it straightforward for DFS to combine multiple small I/O requests to adjacent regions into a single larger I/O. No complicated block layout policies are required at the filesystem layer. This strategy can improve performance because the flash device delivers higher transfer rates with larger I/Os. Our current implementation aggressively merges I/O requests; a more nuanced policy might improve performance further.

DFS leverages the three main operations supported by the virtualized flash storage layer: read from a logical block, write to a logical block, and discard a logical block range. The discard directive marks a logical block range

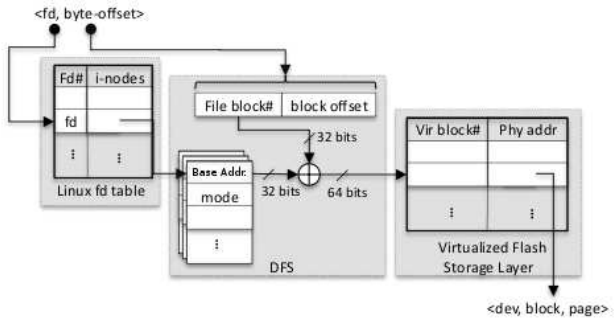


Figure 2: DFS logical block address mapping for large files; only the width of the file block number differs for small files

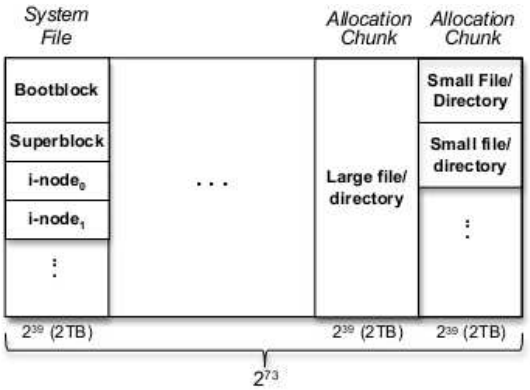


Figure 3: Layout of DFS system and user files in virtualized flash storage. The first 2TB is used for system files. The remaining 2TB allocation chunks are for user data or directory files. A large file takes the whole chunk; multiple small files are packed into a single chunk.

as garbage for the garbage collector and ensures that subsequent reads to the range return only zeros. A version of the discard directive already exists in many flash devices as a hint to the garbage collector; DFS, by contrast, depends upon it to implement truncate and remove. It is also possible to interrogate a logical block range to determine if it contains allocated blocks. The current version of DFS does not make use of this feature, but it could be used by archival programs such as `tar` that have special representations for sparse files.

3.3.2 DFS Layout and Objects

The DFS file system uses a simple approach to store files and their metadata. It divides the 64-bit block addressed virtual flash storage space (DFS volume) into block addressed subspaces or allocation chunks. The size of these two types of subspaces are configured when the filesystem is initialized. DFS places large files in their own allocation chunks and stores multiple small files in a chunk.

As shown in Figure 3, there are three kinds of files in the DFS file system. The first file is a system file which includes the boot block, superblock and all I-nodes. This file is a "large" file and occupies the first allocation chunk at the beginning of the raw device. The boot block occupies the first few blocks (sectors) of the raw device. A superblock immediately follows the boot block. At mount time, the file system can compute the location of the superblock directly. The remainder of the system file contains all I-nodes as an array of block-aligned I-node data structures.

Each I-node is identified by a 32-bit unique identifier or I-node number. Given the I-node number, the logical address of the I-node within the I-node file can be computed directly. Each I-node data structure is stored in a single 512-byte flash block. Each I-node contains the I-number, base virtual address of the corresponding file, mode, link count, file size, user and group IDs, any special flags, a generation count, and access, change, birth, and modification times with nanosecond resolution. These fields take a total of 72 bytes, leaving 440 bytes for additional attributes and future use. Since an I-node fits in a single flash page, it will be updated atomically by the virtualized flash storage layer.

The implementation of DFS uses a 32-bit block-addressed allocation chunk to store the content of a regular file. Since a file is stored in a contiguous, flat space, the address of each block offset can be simply computed by adding the offset to the virtual base address of the space for the file. A block read simply returns the content of the physical flash page mapped to the virtual block. A write operation writes the block to the mapped physical flash page directly. Since the virtualized flash storage layer triggers a mapping or remapping on write, DFS does the write without performing an explicit block allocation. Note that DFS allows holes in a file without using physical flash pages because of the dynamic mapping. When a file is deleted, the DFS will issue a deallocation operation provided by the virtualized flash storage layer to deallocate and unmap virtual space of the entire file.

A DFS directory is mapped to flash storage in the same manner as ordinary files. The only difference is its internal structure. A directory contains contains an array of name, I-node number, type triples. The current implementation is very similar to that found in FFS [22]. Updates to directories, including operations such as rename, which touch multiple directories and the on-flash I-node allocator, are made crash-recoverable through the use of a write-ahead log. Although widely used and simple to implement, this approach does not scale well to large directories. The current version of the virtualized flash storage layer does not export atomic multi-block updates. We anticipate reimplementing directories using hashing and a sparse virtual address space made crash recoverable with atomic updates.

3.3.3 Direct Data Accesses

DFS promotes direct data access. The current Linux implementation of DFS allows the use of the buffer cache in order to support memory mapped I/O which is required for the `exec` system call. However, for many workloads of interest, particularly databases, clients are expected to bypass the buffer cache altogether. The current implementation of DFS provides direct access via the direct I/O buffer cache bypass mechanism already present in the Linux kernel. Using direct I/O, page-aligned reads and writes are converted directly into I/O requests to the block device driver by the kernel.

There are two main rationales for this approach. First, traditional buffer cache design has several drawbacks. The traditional buffer cache typically uses a large amount of memory. Buffer cache design is quite complex since it needs to deal with multiple clients, implement sophisticated cache replacement policies to accommodate various access patterns of different workloads, and maintain consistency between the buffer cache and disk drives, and support crash recovery. In addition, having a buffer cache imposes a memory copy in the storage software stack.

Second, flash memory devices provide low-latency accesses, especially for random reads. Since the virtualized flash storage layer can solve the write latency problem, the main motivation for the buffer cache is largely eliminated. Thus, applications can benefit from the DFS direct data access approach by utilizing most of the main memory space typically used for the buffer cache for a larger in memory working set.

3.3.4 Crash Recovery

The virtualized flash storage layer implements the basic functionality of crash recovery for the mapping from logical block addresses to physical flash storage locations. DFS leverages this property to provide crash recovery. Unlike traditional file systems that use non-volatile random access memory (NVRAM) and their own logging implementation, DFS piggybacks on the flash storage layer's log.

NVRAM and file system level logging require complex implementations and introduce additional costs for the traditional file systems. NVRAM is typically used in high-end file systems so that the file system can achieve low-latency operations while providing fault isolations and avoiding data loss in case of power failures. The traditional logging approach is to log every write and performs group commits to reduce overhead. Logging writes to disk can impose significant overheads. A more efficient approach is to log updates to NVRAM, which is the method typically used in high-end file systems [12]. NVRAMs are typically implemented with battery-backed DRAMs on a PCI card whose price is similar to a few high-density magnetic disk drives. NVRAMs can substantially reduce the file system write performance because every write must go through the NVRAM. For a network file system, each write will have to go through the I/O bus three times, once for the NIC, once for NVRAM, and once for writing to disks.

Since flash memory is a form of NVRAM, DFS leverages the support from the virtualized flash storage layer to achieve crash recoverability. When a DFS file system object is extended, DFS passes the write request to the virtualized flash storage layer which then allocates a physical page of the flash device and logs the result internally. After a crash, the virtualized flash storage layer runs recovery using the internal log. The consistency of the contents of individual files is the responsibility of applications, but the on-flash state of the file system is guaranteed to be consistent. Since the virtualized flash storage layer uses a log-structured approach to tracking allocations for performance reasons and must handle crashes in any case, DFS does not impose any additional onerous requirements.

3.3.5 Discussion

The current DFS implementation has several limitations. The first is that it does not yet support snapshots. One of the reasons we did not implement snapshot is that we plan to support snapshots natively in the virtualized flash storage layer which will greatly simplify the snapshot implementation in DFS. Since the virtualized flash storage layer is already log-structured for performance and hence takes a copy-on-write approach by default, one can implement snapshots in the virtualized flash storage layer efficiently.

The second is that we are currently implementing support for atomic multi-block updates in the virtualized flash storage layer. The log-structured, copy-on-write nature of the flash storage layer makes it possible to export such an interface efficiently. For example, Prabhakaran *et al.* recently described an efficient commit protocol to implement atomic multi-block writes [25]. This type of methods will allow DFS to guarantee the consistency of directory contents and I-node allocations in a simple fashion. In the interim, DFS uses a straightforward extension of the traditional UFS/FFS directory structure.

The third is the limitations on the number of files and the maximum file size. We have considered a design that supports two file sizes: small and very large. The file layout algorithm initially assumes a file is small (*e.g.,* less than 2GB). If it needs to exceed the limit, it will become a very large file (*e.g.,* up to 2PB). The virtual block address space is partitioned so that a large number of small file ranges are mapped in one partition and a smaller number of very large file ranges are mapped into the remaining partition. A file may be promoted from the small partition to the very large partition by copying the mapping of a

virtual flash storage address space to another at the virtualized flash storage layer. We plan to export such support and implement this design in the next version of DFS.

4 Evaluation

We are interested in answering two main questions:

- How do the layers of abstraction perform?
- How does DFS compare with existing file systems?

To answer the first question, we use a microbenchmark to evaluate the number of I/O operations per second (IOPS) and bandwidth delivered by the virtualized flash storage layer and by the DFS layer. To answer the second question, we compare DFS with ext3 by using a microbenchmark and an application suite. Ideally, we would compare with existing flash filesystems as well, however filesystems such as YAFFS and JFFS2 are designed to use raw NAND flash and are not compatible with next-generation flash storage that exports a block interface.

All of our experiments were conducted on a desktop with Intel Quad Core processor running at 2.4GHz with a 4MB cache and 4GB DRAM. The host operating system was a stock Fedora Core installation running the Linux 2.6.27.9 kernel. Both DFS and the virtualized flash storage layer implemented by the FusionIO device driver were compiled as loadable kernel modules.

We used a FusionIO ioDrive with 160GB of SLC NAND flash connected via PCI-Express x4 [1]. The advertised read latency of the FusionIO device is $50\mu s$. For a single reader, this translates to a theoretical maximum throughput of 20,000 IOPS. Multiple readers can take advantage of the hardware parallelism in the device to achieve much higher aggregate throughput. For the sake of comparison, we also ran the microbenchmarks on a 32GB Intel X25-E SSD connected to a SATA II host bus adapter [2]. This device has an advertised typical read latency of about $75\mu s$.

Our results show that the virtualized flash storage layer delivers performance close to the limits of the hardware, both in terms of IOPS and bandwidth. Our results also show that DFS is much simpler than ext3 and achieves better performance in both the micro- and application benchmarks than ext3, often using less CPU power.

4.1 Virtualized Flash Storage Performance

We have two goals in evaluating the performance of the virtualized flash storage layer. First, to examine the potential benefits of the proposed abstraction layer in combination with hardware support that exposes parallelism. Second, to determine the raw performance in terms of bandwidth and IOPs delivered in order to compare DFS and ext3. For both purposes, we designed a simple microbenchmark which opens the raw block device in direct I/O mode, bypassing the kernel buffer cache. Each thread in the program attempts to execute block-aligned reads and writes as quickly as possible.

To evaluate the benefits of the virtualized flash storage layer and its hardware, one would need to compare a traditional block storage software layer with flash memory hardware equivalent to the FusionIO ioDrive but with a traditional disk interface FTL. Since such hardware does not exist, we have used a Linux block storage layer with an Intel X25-E SSD, which is a well-regarded SSD in the marketplace. Although this is not a fair comparison, the results give us some sense of the performance impact of the abstractions designed for flash memory.

We measured the number of sustained random I/O transactions per second. While both flash devices are enterprise class devices, the test platform is the typical white box workstation we described earlier. The results are shown in Figure 4. Performance, while impressive compared to magnetic disks, is less than that advertised by the manufacturers. We suspect that the large IOPS performance gaps, particularly for write IOPS, are partially limited by the disk drive interface and limited resources in a drive controller to run sophisticated remapping algorithms.

Device	Read IOPS	Write IOPS
Intel	33,400	3,120
FusionIO	98,800	75,100

Figure 4: Device 4KB Peak Random IOPS

Device	Threads	Read (MB/s)	Write (MB/s)
Intel	2	221	162
FusionIO	2	769	686

Figure 5: Device Peak Bandwidth 1MB Transfers

Figure 5 shows the peak bandwidth for both cases. We measured sequential I/O bandwidth by computing the aggregate throughput of multiple readers and writers. Each client transferred 1MB blocks for the throughput test and used direct I/O to bypass the kernel buffer cache. The results in the table are the bandwidth results using two writers. The virtualized flash storage layer with ioDrive achieves 769MB/s for read and 686MB/s for write, whereas the traditional block storage layer with the Intel SSD achieves 221MB/s for read and 162MB/s for write.

4.2 Complexity of DFS vs. ext3

Figure 6 shows the number of lines of code for the major modules of DFS and ext3 file systems. Although both implement Unix file systems, DFS is much simpler. The

Module	DFS	Ext3
Headers	392	1583
Kernel Interface (Superblock, *etc.*)	1625	2973
Logging	0	7128
Block Allocator	0	1909
I-nodes	250	6544
Files	286	283
Directories	561	670
ACLs, Extended Attrs.	N/A	2420
Resizing	N/A	1085
Miscellaneous	175	113
Total	3289	24708

Figure 6: Lines of Code in DFS and Ext3 by Module

simplicity of DFS is mainly due to delegating block allocations and reclamations to the virtualized flash storage layer. The ext3 file system, for example, has a total of 17,500 lines of code and relies on an additional 7,000 lines of code to implement logging (JBD) for a total of nearly 25,000 lines of code compared to roughly 3,300 lines of code in DFS. Of the total lines in ext3, about 8,000 lines (33%) are related to block allocations, deallocations and I-node layout. Of the remainder, another 3,500 lines (15%) implement support for on-line resizing and extended attributes, neither of which are supported by DFS.

Although it may not be fair to compare a research prototype file system with a file system that has evolved for several years, the percentages of block allocation and logging in the file systems give us some indication of the relative complexity of different components in a file system.

4.3 Microbenchmark Performance of DFS vs. ext3

We use Iozone [23] to evaluate the performance of DFS and ext3 on the ioDrive when using both direct and buffered access. We record the number of 4KB I/O transactions per second achieved with each file system and also compute the CPU usage required in each case as the ratio between user plus system time to elapsed wall time. For both file systems, we ran Iozone in three different modes: in the default mode in which I/O requests pass through the kernel buffer cache, in direct I/O mode without the buffer cache, and in memory-mapped mode using the `mmap` system call.

In our experiments, both file systems run on top of the virtualized flash storage layer. The ext3 file system in this case uses the backward compatible block storage interface supported by the virtualized flash storage layer.

Direct Access

For both reads and writes, we consider sequential and uniform random access to previously allocated blocks. Our goal is to understand the additional overhead due to DFS compared to the virtualized flash storage layer. The results indicate that DFS is indeed lightweight and imposes much less overhead than ext3. Compared to the raw device, DFS delivers about 5% fewer IOPS for both read and write whereas ext3 delivers 9% fewer read IOPS and more than 20% fewer write IOPS. In terms of bandwidth, DFS delivers about 3% less write bandwidth whereas ext3 delivers 9% less write bandwidth.

File System	Threads	Read (MB/s)	Write (MB/s)
ext3	2	760	626
DFS	2	769	667

Figure 7: Peak Bandwidth 1MB Transfers on ioDrive

Figure 7 shows the peak bandwidth for sequential 1MB block transfers. This microbenchmark is the filesystem analog of the raw device bandwidth performance shown in Figure 5. Although the performance difference between DFS and ext3 for large block transfers is relatively modest, DFS does narrow the gap between filesystem and raw device performance for both sequential reads and writes.

Figure 8 shows the average direct random I/O performance on DFS and ext3 as a function of the number of concurrent clients on the FusionIO ioDrive. Both of the file systems also exhibit a characteristic that may at first seem surprising: *aggregate* performance often increases with an increasing number of clients, even if the client requests are independent and distributed uniformly at random. This behavior is due to the relatively long latency of individual I/O transactions and deep hardware and software request queues in the flash storage subsystem. This behavior is quite different from what most applications expect and may require changes to them in order to realize the full potential of the storage system.

Unlike read throughput, write throughput peaks at about 16 concurrent writers and then decreases slightly. Both the aggregate throughput and the number of concurrent writers at peak performance are lower than when accessing the raw storage device. The additional overhead imposed by the filesystem on the write path reduces both the total aggregate performance and the number of concurrent writers that can be handled efficiently.

We have also measured CPU utilization per 1,000 IOPS delivered in the microbenchmarks. Figure 9 shows the improvement of DFS over ext3. We report the average of five runs of the IOZone based microbenchmark with a standard deviation of one to three percent. For reads, DFS CPU utilization is comparable to ext3; for writes, particularly with small numbers of threads, DFS is more efficient. Overall, DFS consumes somewhat less CPU power, further confirming that DFS is a lighter weight file system than ext3.

One anomaly worthy of note is that DFS is actually

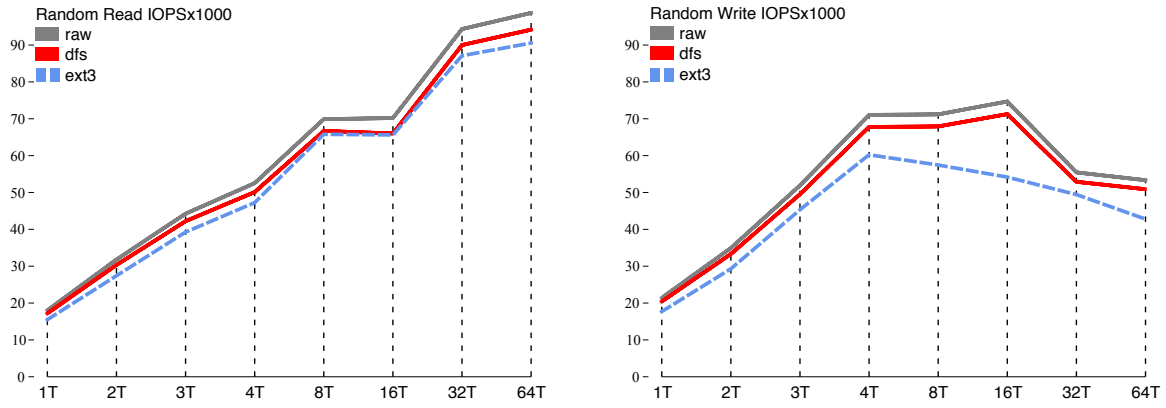


Figure 8: Aggregate IOPS for 4K Random Direct I/O as a Function of the Number of Threads

Threads	Read	Random Read	Write	Random Write
1	8.1	2.8	9.4	13.8
2	1.3	1.6	12.8	11.5
3	0.4	5.8	10.4	15.3
4	-1.3	-6.8	-15.5	-17.1
8	0.3	-1.0	-3.9	-1.2
16	1.0	1.7	2.0	6.7
32	4.1	8.5	4.8	4.4

Figure 9: Improvement in CPU Utilization per 1,000 IOPS using 4K Direct I/O with DFS relative to Ext3

more expensive than ext3 per I/O when running with four clients, particularly if the clients are writers. This is due to the fact that there are four cores on the test machine and the device driver itself has worker threads that require CPU and memory bandwidth. The higher performance of DFS translates into more work for the device driver and particularly for the garbage collector. Since there are more threads than cores, cache hit rates suffer and scheduling costs increase; under higher offered load, the effect is more pronounced, although it can be mitigated somewhat by binding the garbage collector to a single processor core.

Buffered Access

To evaluate the performance of DFS in the presence of the kernel buffer cache, we ran a similar set of experiments as in the case of direct I/O. Each experiment touched 8GB worth of data using 4K block transfers. The buffer cache was invalidated after each run by unmounting the file system and the total data referenced exceeded the physical memory available by a factor of two. The first run of each experiment was discarded and the average of the subsequent ten runs reported.

Figures 10 and 11 show the results via the Linux buffer cache and via memory-mapped I/O data path which also uses the buffer cache. There are several observations.

Thr.	Seq. Read IOPS x 1K ext3	Seq. Read IOPS x 1K DFS (Speedup)	Rand. Read IOPS x 1K ext3	Rand. Read IOPS x 1K DFS (Speedup)
1	125.5	191.2 (1.52)	17.5	19.0 (1.09)
2	147.6	194.1 (1.32)	32.9	34.0 (1.03)
3	137.1	192.7 (1.41)	44.3	46.6 (1.05)
4	133.6	193.9 (1.45)	55.2	57.8 (1.05)
8	134.4	193.5 (1.44)	78.7	80.5 (1.02)
16	132.6	193.9 (1.46)	79.6	81.1 (1.02)
32	132.3	194.8 (1.47)	95.4	101.2 (1.06)
Thr.	Seq. Write IOPS x 1K ext3	Seq. Write IOPS x 1K DFS (Speedup)	Rand. Write IOPS x 1K ext3	Rand. Write IOPS x 1K DFS (Speedup)
1	67.8	154.9 (2.28)	61.2	68.5 (1.12)
2	71.6	165.6 (2.31)	56.7	64.6 (1.14)
3	73.0	156.9 (2.15)	59.6	62.8 (1.05)
4	65.5	161.5 (2.47)	57.5	63.3 (1.10)
8	64.9	148.1 (2.28)	57.0	58.7 (1.03)
16	65.3	147.8 (2.26)	52.6	56.5 (1.07)
32	65.3	150.1 (2.30)	55.2	50.6 (0.92)

Figure 10: Buffer Cache Performance with 4KB I/Os

First, both DFS and ext3 have similar random read IOPS and random write IOPS to their performance results using direct I/O. Although this is expected, DFS is better than ext3 on average by about 5%. This further shows that DFS has less overhead than ext3 in the presence of a buffer cache.

Second, we observe that the traditional buffer cache is not effective when there are a lot of parallel accesses. In the sequential read case, the number of IOPS delivered by DFS basically doubles its direct I/O access performance, whereas the IOPS of ext3 is only modestly better than its random access performance when there are enough parallel accesses. For example, when there are 32 threads, its IOPS is 132,000, which is only 28% better than its random read IOPS of 95,400!

Third, DFS is substantially better than ext3 for both sequential read and sequential write cases. For sequential reads, it outperforms ext3 by more than a factor of 1.4. For sequential writes, it outperforms ext3 by more than a

Thr.	Seq. Read IOPS x 1K ext3	Seq. Read IOPS x 1K DFS (Speedup)	Rand. Read IOPS x 1K ext3	Rand. Read IOPS x 1K DFS (Speedup)
1	42.6	52.2 (1.23)	13.9	18.1 (1.3)
2	72.6	84.6 (1.17)	22.2	28.2 (1.27)
3	94.7	114.9 (1.21)	27.4	32.1 (1.17)
4	110.2	117.1 (1.06)	29.7	35.0 (1.18)
Thr.	Seq. Write IOPS x 1K ext3	Seq. Write IOPS x 1K DFS (Speedup)	Rand. Write IOPS x 1K ext3	Rand. Write IOPS x 1K DFS (Speedup)
1	28.8	40.2 (1.4)	11.8	13.5 (1.14)
2	39.9	55.5 (1.4)	16.7	18.1 (1.08)
3	41.9	68.4 (1.6)	19.1	20.0 (1.05)
4	44.3	70.8 (1.6)	20.1	22.0 (1.09)

Figure 11: Memory Mapped Performance of Ext3 & DFS

factor of 2.15. This is largely due to the fact that DFS is simple and can easily combines I/Os.

The story for memory-mapped I/O performance is much the same as it is for buffered I/O. Random access performance is relatively poor compared to direct I/O performance. The simplicity of DFS and the short code paths in the filesystem allow it to outperform ext3 in this case. The comparatively large speedups for sequential I/O, particularly sequential writes, is again due to the fact that DFS readily combines multiple small I/Os into larger ones. In the next section we show that I/O combining is an important effect; the quicksort benchmark is a good example of this phenomenon with memory mapped I/O. We count both the number of I/O transactions during the course of execution and the total number of bytes transferred. DFS greatly reduces the number of write operations and more modestly the number of read operations.

4.4 Application Benchmarks Performance of DFS vs. ext3

We have used five applications as an application benchmark suite to evaluate the application-level performance on DFS and ext3.

Application Benchmarks

The table in Figure 12 summarizes the characteristics of the applications and the reasons why they are chosen for our performance evaluation.

In the following, we describe each application, its implementation and workloads in detail:

Quicksort. This quicksort is implemented as a single-threaded program to sort 715 million 24 byte key-value pairs memory mapped from a single 16GB file. Although quicksort exhibits good locality of reference, this benchmark program nonetheless stresses the memory mapped I/O subsystem. The memory-mapped interface has the advantages of being simple, easy to understand, and a straightforward way to transform a large flash storage device into an inexpensive replacement for DRAM as it provides the illusion of word-addressable access.

Applications	Description	I/O Patterns
Quicksort	A quicksort on a large dataset	Mem-mapped I/O
N-Gram	A program for querying n-gram data	Direct, random read
KNNImpute	Processes bioinformatics microarray data	Mem-mapped I/O
VM-Update	Update of an OS on several virtual machines	Sequential read & write
TPC-H	Standard benchmark for Decision Support	Mostly sequential read

Figure 12: Applications and their characteristics.

N-Gram. This program indexes all of the 5-grams in the Google n-gram corpus by building a single large hash table that contains 26GB worth of key-value pairs. The Google n-gram corpus is a large set of n-grams and their appearance counts taken from a crawl of the Web that has proved valuable for a variety of computational linguistics tasks. There are just over 13.5 million words or 1-grams and just over 1.1 billion 5 grams. Indexing the data set with an SQL database takes a week on a computer with only 4GB of DRAM [9]. Our indexing program uses 4KB buckets with the first 64 bytes reserved for metadata. The implementation does not support overflows, rather an occupancy histogram is constructed to find the smallest k such that 2^k hash buckets will hold the dataset without overflows. With a variant of the standard Fowler-Nolls-Vo hash, the entire data set fits in 16M buckets and the histogram in 64MB of memory. Our evaluation program uses synthetically generated query traces of 200K queries each; results are based upon the average of twenty runs. Queries are drawn either uniformly at random or according to a Zipf distribution with $\alpha = 1.0001$. The results were qualitatively similar for other values of α until locking overhead dominated I/O overhead.

KNNImpute. This program is a very popular bionformatics code for estimating missing values in data obtained from microarray experiments. The program uses the KNNImpute [29] algorithm for DNA microarrays which takes as input a matrix with G rows representing genes and E columns representing experiments. Then a symmetric GxG distance matrix with the Euclidean distance between all gene pairs is calculated based on all experiment values for both genes. Finally, the distance matrix is written to disk as its output. The program is a multi-threaded implementation using memory-mapped I/O. Our input data is a matrix with 41,768 genes and 200 experiments results in a matrix of about 32MB, and a distance matrix of 6.6GB. There are 2079 genes with missing values.

VM Update. This benchmark is a simple update of multiple virtual machines hosted on a single server. We

Application	Wall Time		
	Ext3	DFS	Speedup
Quick Sort	1268	822	1.54
N-Gram (Zipf)	4718	1912	2.47
KNNImpute	303	248	1.22
VM Update	685	640	1.07
TPC-H	5059	4154	1.22

Figure 13: Application Benchmark Execution Time Improvement: Best of DFS *vs* Best of Ext3

choose this application because virtual machines have become popular from both a cost and management perspective. Since each virtual machine typically runs the same operating system but has its own copy, operating system updates can pose a significant performance problem. Each virtual machine needs to apply critical and periodic system software updates at the same time. This process is both CPU and I/O intensive. To simulate such an environment, we installed 4 copies of Ubuntu 8.04 in four different VirtualBox instances. In each image, we downloaded all of the available updates and then measured the amount of time it took to install these updates. There were a total of 265 packages updated containing 343MB of compressed data and about 38,000 distinct files.

TPC-H. This is a standard benchmark for decision support workloads. We used the Ingres database to run the Transaction Processing Council's Benchmark H (TPC-H) [4]. The benchmark consists of 22 business oriented queries and two functions that respectively insert and delete rows in the database. We used the default configuration for the database with two storage devices: the database itself, temporary files, and backup transaction log were placed on the flash device and the executables and log files were stored on the local disk. We report the results of running TPC-H with a scale factor of 5, which corresponds to about 5GB of raw input data and 90GB for the data, indexes, and logs stored on flash once loaded into the database.

Performance Results of DFS vs. ext3

This section first reports the performance results of DFS and ext3 for each application, and then analyzes the results in detail.

The main performance result is that DFS improves applications substantially over ext3. Figures 13 shows the elapsed wall time of each application running with ext3 and DFS on the same execution environment mentioned at the beginning of the section. The results show that DFS improves the performance all applications and the speedups range from a factor of 1.07 to 2.47.

To explain the performance results, we will first use Figure 14 to show the number of read and write IOPS, and the number of bytes transferred for reads and writes for each application. The main observation is that DFS issues a smaller number of larger I/O transactions than ext3, though the behaviors of reads and writes are quite different. This observation explains partially why DFS improves the performance of all applications, since we know from the microbenchmark performance that DFS achieves better IOPS than ext3 and significantly better throughput when the I/O transaction sizes are large.

One reason for larger I/O transactions is that in the Linux kernel, file offsets are mapped to block numbers via a per-file-system `get_block` function. The DFS implementation of `get_block` is aggressive about making large transfers when possible. A more nuanced policy might improve performance further, particularly in the case of applications such as KNNImpute and the VM Update workload which actually see an increase in the total number of bytes transferred. In most cases, however, the result of the current implementation is a modest reduction in the number of bytes transferred.

But, the smaller number of larger I/O transactions does not completely explain the performance results. In the following, we will describe our understanding of the performance of each application individually.

Quicksort. The Quicksort benchmark program sees a speedup of 1.54 when using DFS instead of ext3 on the ioDrive. Unlike the other benchmark applications, the quicksort program sees a large increase in CPU utilization when using DFS instead of ext3. CPU utilization includes both the CPU used by the FusionIO device driver and by the application itself. When running on ext3, this benchmark program is I/O bound; the higher throughput provided by DFS leads to higher CPU utilization, which is actually a desirable outcome in this particular case. In addition, we collected statistics from the virtualized flash storage layer to count the number of read and write transactions issued in each of the three cases. When running on ext3, the number of read transactions is similar to that found with DFS, whereas the number of write transactions is roughly twenty-five times larger than that of DFS, which contributed to the speedup. The average transaction size with ext3 is about 4KB instead of 64KB with DFS.

Google N-Gram Corpus. The N-gram query benchmark program running on DFS achieves a speedup of 2.5 over that on ext3. Figure 15 illustrates the speedup as a function of the number of concurrent threads; in all cases, the internal cache is 1,024 hash buckets and all I/O bypasses the kernel's buffer cache.

The hash table implementation is able to achieve about 95% of the random I/O performance delivered in the `Iozone` microbenchmarks given sufficient concurrency. As expected, performance is higher when the queries are Zipf-distributed as the internal cache captures many of the most popular queries. For Zipf parameter $\alpha = 1.0001$, there are about 156,000 4K random reads to sat-

Application	Read IOPS x 1000		Read Bytes x 1M		Write IOPS x 1000		Write Bytes x 1M	
	Ext3	DFS (Change)	Ext3	DFS (Change)	Ext3	DFS (Change)	Ext3	DFS (Change)
Quick Sort	1989	1558 (0.78)	114614	103991 (0.91)	49576	1914 (0.04)	203063	192557 (0.95)
N-Gram (Zipf)	156	157 (1.01)	641	646 (1.01)	N/A	N/A	N/A	N/A
KNNImpute	2387	1916 (0.80)	42806	36146 (0.84)	2686	179 (0.07)	11002	12696 (1.15)
VM Update	244	193 (0.79)	9930	9760 (0.98)	3712	1144 (0.31)	15205	19767 (1.30)
TPC-H	6375	3760 (0.59)	541060	484985 (0.90)	52310	3626 (0.07)	214265	212223 (0.99)

Figure 14: App. Benchmark Improvement in IOPS Required and Bytes Transferred: Best of DFS *vs* Best of Ext3

Threads	Wall Time in Sec.		Ctx Switch x 1K	
	Ext3	DFS	Ext3	DFS
1	10.82	10.48	156.66	156.65
4	4.25	3.40	308.08	160.60
8	4.58	2.46	291.91	167.36
16	4.65	2.45	295.02	168.57
32	4.72	1.91	299.73	172.34

Figure 15: Zipf-Distributed N-Gram Queries: Elapsed Time and Context Switches ($\alpha = 1.0001$)

isfy 200,000 queries. Moreover, query performance for hash tables backed by DFS scales with the number of concurrent threads much as it did in the `Iozone` random read benchmark. The performance of hash tables backed by ext3 do not scale with the number of threads nearly so well. This is due to increased per-file lock contention in ext3. We measured the number of voluntary context switches when running on each file system as reported by `getrusage`. A voluntary context switch indicates that the application was unable to acquire a resource in the kernel such as a lock. When running on ext3, the number of voluntary context switches increased dramatically with the number of concurrent threads; it did not do so on DFS. Although it may be possible to overcome the resource contention in ext3, the simplicity of DFS allows us to sidestep the issue altogether. This effect was less pronounced in the microbenchmarks because `Iozone` never assigns more than one thread to each file by default.

Bioinformatics Missing Value Estimation. KNNImpute takes about 18% less time to run when using DFS as opposed to ext3 with a standard deviation of about 1% of the mean run time. About 36% of the total execution time when running on ext3 is devoted to writing the distance matrix to stable storage. Most of the improvement in run time when running on DFS is during this phase of execution. CPU utilization increases by almost 7% on average when using DFS instead of ext3. This is due to increased system CPU usage during the distance matrix write phase by the FusionIO device driver's worker threads, particularly the garbage collector.

Virtual Machine Update. On average, it took 648 seconds to upgrade virtual machines hosted on DFS and 701 seconds to upgrade those hosted on ext3 file systems, for a net speed up of 7.6%. In both cases, the four virtual machines used nearly all of the available CPU for the duration of the benchmark. We found that each VirtualBox instance kept a single processor busy almost 25% percent of the time even when the guest operating system was idle. As a result, the virtual machine update workload quickly became CPU bound. If the virtual machine implementation itself were more efficient or more virtual machines shared the same storage system we would expect to see a larger benefit to using DFS.

TPC-H. We ran the TPC-H benchmark with a scale factor of five on both DFS and ext3. The average speedup over five runs was 1.22. For the individual queries DFS always performs better than ext3, with the speedup ranging from 1.04 (Q1: pricing summary report) to 1.51 (RF2: old sales refresh function). However, the largest contribution to the overall speedup is the 1.20 speedup achieved for Q5 (local supplier volume), which consumes roughly 75% of the total execution time.

There is a large reduction (14.4x) in the number of write transactions when using DFS as compared to ext3 and a smaller reduction (1.7x) in the number of read transactions. As in the case of several of the other benchmark applications, the large reduction in the number of I/O transactions is largely offset by larger transfers in each transaction, resulting in a modest decrease in the total number of bytes transferred.

CPU utilization is lower when running on DFS as opposed to ext3, but the Ingres database thread runs with close to 100% CPU utilization in both cases. The reduction in CPU usage is due instead to greater efficiency in the kernel storage software stack, particularly the flash device driver's worker threads.

5 Conclusion

This paper presents the design, implementation, and evaluation of DFS and describes FusionIO's virtualized flash storage layer. We have demonstrated that novel layers of abstraction specifically for flash memory can yield substantial benefits in software simplicity and system performance.

We have learned several things from DFS design. First, DFS is simple and has a short and direct way to access flash memory. Much of its simplicity comes from leveraging the virtualized flash storage layer such as large virtual storage space, block allocation and deallocation, and

atomic block updates.

Second, the simplicity of DFS translates into performance. Our microbenchmark results show that DFS can deliver 94,000 IOPS for random reads and 71,000 IOPS random writes with the virtualized flash storage layer on FusionIO's ioDrive. The performance is close to the hardware limit.

Third, DFS is substantially faster than ext3. For direct access performance, DFS is consistently faster than ext3 on the same platform, sometimes by 20%. For buffered access performance, DFS is also consistently faster than ext3, and sometimes by over 149%. Our application benchmarks show that DFS outperforms ext3 by 7% to 250% while requiring less CPU power.

We have also observed that the impact of the traditional buffer cache diminishes when using flash memory. When there are 32 threads, the sequential read throughput of DFS is about twice that for direct random reads with DFS, whereas ext3 achieves only a 28% improvement over direct random reads with ext3.

References

[1] FusionIO ioDrive specification sheet. `http://www.fusionio.com/PDFs/Fusion%20Specsheet.pdf`.

[2] Intel X25-E SATA solid state drive. `http://download.intel.com/design/flash/nand/extreme/extreme-sata-ssd-datasheet.pdf`.

[3] Understanding the flash translation layer (FTL) specification. Tech. Rep. AP-684, Intel Corporation, December 1998.

[4] *TPC Benchmark H: Decision Support*. Transaction Processing Performance Council (TPC), 2008. `http://www.tpc.org/tpch`.

[5] AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J. D., MANASSE, M., AND PANIGRAHY, R. Design tradeoffs for SSD performance. In *Proceedings of the 2008 USENIX Technical Conference* (June 2008).

[6] BIRRELL, A., ISARD, M., THACKER, C., AND WOBBER, T. A design for high-performance flash disks. *ACM Operating Systems Review 41*, 2 (April 2007).

[7] BRANTS, T., AND FRANZ, A. Web 1T 5-gram version 1, 2006.

[8] CARD, R., T'SO, T., AND TWEEDIE, S. The design and implementation of the second extended filesystem. In *First Dutch International Symposium on Linux* (December 1994).

[9] CARLSON, A., MITCHELL, T. M., AND FETTE, I. Data analysis project: Leveraging massive textual corpora using n-gram statistics. Tech. Rep. CMU-ML-08-107, Carnegie Mellon University Machine Learning Department, May 2008.

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

[11] DOUGLIS, F., CACERES, R., KAASHOEK, M. F., LI, K., MARSH, B., AND TAUBER, J. A. Storage alternatives for mobile computers. In *Operating Systems Design and Implementation* (1994), pp. 25–37.

[12] HITZ, D., LAU, J., AND MALCOM, M. File system design for an nfs file server appliance. Tech. Rep. TR-3002, NetApp Corporation, September 2001.

[13] JO, H., KANG, J.-U., PARK, S.-Y., KIM, J.-S., AND LEE, K. FAB: Flash-aware buffer management policy for portable media players. *IEEE Transactions on Consumer Electronics 52*, 2 (2006), 485–493.

[14] KAWAGUCHI, A., NISHIOKA, S., AND MOTODA, H. A flash-memory based file system. In *In Proceedings of the Winter 1995 USENIX Technical Conference* (1995).

[15] KIM, H., AND AHN, S. BPLRU: A buffer management scheme for improving random writes in flash storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies* (February 2008).

[16] KIM, J., KIM, J. M., NOH, S. H., MIN, S. L., , AND CHO, Y. A space-efficient flash translation layer for CompactFlash systems. *IEEE Transactions on Consumer Electronics 48*, 2 (2002), 366–375.

[17] LI, K. Towards a low power file system. Tech. Rep. CSD-94-814, University of California at Berkeley, May 1994.

[18] LLANOS, D. R. TPCC-UVa: An open-source TPC-C implementation for global performance measurement of comptuer systems. *ACM SIGMOD Record 35*, 4 (December 2006), 6–15.

[19] MANNING, C. YAFFS: The NAND-specific flash file system. *LinuxDevices.Org* (September 2002).

[20] MARSH, B., DOUGLIS, F., AND KRISHNAN, P. Flash memory file caching for mobile computers. In *Proceedings of the Twenty-Seventh Hawaii Internanal Conference on Architecture* (January 1994).

[21] MATHUR, A., CAO, M., BHATTACHARYA, S., DILGER, A., TOMAS, A., AND VIVIER, L. The new ext4 filesystem: Current status and future plans. In *Ottowa Linux Symposium* (June 2007).

[22] MCKUSICK, M. K., JOY, W. N., LEFFLER, S. J., AND FABRY, R. S. A fast file system for UNIX. *ACM Transactions on Computer Systems 2*, 3 (August 1984).

[23] NORCOTT, W. Iozone filesystem benchmark. `http://www.iozone.org`.

[24] PARK, S.-Y., JUNG, D., KANG, J.-U., KIM, J.-S., AND LEE, J. CFLRU: A replacement algorithm for flash memory. In *Procieedings of the 2006 International Conference on Compilers, Architecture and Syntehsis for embedded Systems* (2006).

[25] PRABHAKARAN, V., RODEHEFFER, T. L., AND ZHOU, L. Transactional flash. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation* (December 2008).

[26] RAJIMWALE, A., PRABHAKARAN, V., AND DAVIS, J. D. Block management in solid state devices. Unpublished Technical Report, January 2009.

[27] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10* (1992), 1–15.

[28] TANENBAUM, A. S., HERDER, J. N., AND BOS, H. File size distribution in UNIX systems: Then and now. *ACMSIGOPS Operating Systems Review 40*, 1 (January 2006), 100–104.

[29] TROYANSKAYA, O., CANTOR, M., SHERLOCK, G., BROWN, P., HASTIEEVOR, T., TIBSHIRANI, R., BOTSTEIN, D., AND ALTMAN, R. B. Missing value estimation methods for DNA microarrays. *Bioinformatics 17*, 6 (2001), 520–525.

[30] TWEEDIE, S. Ext3, journaling filesystem. In *Ottowa Linux Symposium* (July 2000).

[31] ULMER, C., AND GOKHALE, M. Threading opportunities in high-performance flash-memory storage. In *High Performance Embedded Computing* (2008).

[32] WOODHOUSE, D. JFFS: The journalling flash file system. In *Ottowa Linux Symposium* (2001).

[33] WU, M., AND ZWAENEPOEL, W. eNVy: A non-volatile, main memory storage system. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems* (1994).

Extending SSD Lifetimes with Disk-Based Write Caches

Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, Ted Wobber*
University of Toronto, Microsoft Research Silicon Valley*
gokul@eecg.toronto.edu, {vijayanp, maheshba, wobber}@microsoft.com

Abstract

We present Griffin, a hybrid storage device that uses a hard disk drive (HDD) as a write cache for a Solid State Device (SSD). Griffin is motivated by two observations: First, HDDs can match the sequential write bandwidth of mid-range SSDs. Second, both server and desktop workloads contain a significant fraction of block overwrites. By maintaining a log-structured HDD cache and migrating cached data periodically, Griffin reduces writes to the SSD while retaining its excellent performance. We evaluate Griffin using a variety of I/O traces from Windows systems and show that it extends SSD lifetime by a factor of two and reduces average I/O latency by 56%.

1 Introduction

Over the past decade, the use of flash memory has evolved from specialized applications in hand-held devices to primary system storage in general-purpose computers. Flash-based Solid State Devices (SSDs) provide 1000s of low-latency IOPS and can potentially eliminate I/O bottlenecks in current systems. The cost of commodity flash – often cited as the primary barrier to SSD deployment [22] – has dropped significantly in the recent past, creating the possibility for widespread replacement of disk drives by SSDs.

However, two trends have a potential to derail the adoption of SSDs. First, general-purpose (OS) workloads are harder on the storage subsystem than hand-held applications, particularly in terms of write volume and non-sequentiality. Second, as the cost of NAND flash has declined with increased bit density, the number of erase cycles (and hence write operations) a flash cell can tolerate has suffered. This combination of a more stressful workload and fewer available erase cycles reduces useful lifetime, in some cases to less than one year.

In this paper, we propose Griffin, a hybrid storage design that, somewhat contrary to intuition, uses a hard disk drive to cache writes to an SSD. Writes to Griffin are logged sequentially to the HDD write cache and later migrated to the SSD. Reads are usually served from the SSD and occasionally from the slower HDD. Griffin's goal is to minimize the writes sent to the SSD without significantly impacting its read performance; by doing so, it conserves erase cycles and extends SSD lifetime.

Griffin's hybrid design is based on two characteristics observed in block-level traces collected from systems running Microsoft Windows. First, many of the writes seen by block devices are in fact *overwrites* of a small set of popular blocks. Using an HDD as a write cache to coalesce overwrites can reduce the write traffic to the SSD significantly; for the desktop and server traces we examined, it does by an average of 52%. Second, once data is written to a block device, it is not read again from the device immediately; the file system cache serves any immediate reads without accessing the device. Accordingly, Griffin has a time window within which to coalesce overwrites on the HDD, during which few reads occur.

A log structured HDD makes for an unconventional write cache: writes are fast whereas random reads are slow and can affect the logging bandwidth. By logging writes to the HDD, Griffin takes advantage of the fact that a commodity SATA disk drive delivers over 80 MB/s of sequential write bandwidth, allowing it to keep up with mid-range SSDs. In addition, hard disks offer massive capacity, allowing Griffin to log writes for long periods without running out of space. Since hard disks are very inexpensive, the cost of the write cache is a fraction of the SSD cost.

We evaluate Griffin using a simulator and a user-level implementation with a variety of I/O traces, both from desktop and server environments. Our evaluation shows that, for the desktop workloads we studied, our caching policies can cut down writes to the SSD by approximately 49% on average, with less than 1% of reads serviced by the slower HDD. For server workloads, the observed benefit is more widely varied, but equally signifi-

cant. In addition, Griffin improves the sequentiality of the write accesses to the SSD by an average of 15%, which can indirectly improve the lifetime of the SSD. Reducing the volume of writes by half allows Griffin to extend SSD lifetime by at least a factor of two; by additionally improving the sequentiality of the workload seen by the SSD, Griffin can extend SSD lifetime even more, depending on the SSD firmware design. An evaluation of the performance of Griffin shows that it performs much better than a regular SSD, where the average I/O latency is reduced by 56%.

2 SSD Write-Lifetime

Constraints on the amount of data that can be written to an SSD stem from the properties of NAND flash. Specifically, a block must be erased before being re-written, and only a finite number of erasures are possible before the bit error rate of the device becomes unacceptably high [7, 20]. SLC (single-level cell) flash typically supports 100K erasures per flash block. However, as SSD technology moves towards MLC (multi-level cell) flash that provides higher bit densities at lower cost, the erasure limit per block drops as low as 5,000 to 10,000 cycles. Given that smaller chip feature sizes and more bits-per-cell both increase the likelihood of errors, we can expect erasure limits to drop further as densities increase.

Accordingly, we define a device *write-lifetime*, which is the total number of writes that can be issued to the device over its lifetime. For example, an SSD with 60 GB of NAND flash with 5000 erase-cycles per block might support a maximum write-lifetime of 300 TB (5000 $\times$ 60 GB). However, write-lifetime is unlikely to be optimal in practice, depending on the workload and firmware. For example, according to Micron's data sheet [18], under a specific workload, its 60 GB SSD only has write-lifetime of 42 TB, which is a reduction in write-lifetime by a factor of 7. It is conceivable that under a more stressful workload, SSD write-lifetime decreases by more than an order of magnitude.

Firmware on commodity SSDs can reduce write-lifetime due to inefficiencies in the Flash Translation Layer (FTL), which maintains a map between host logical sector addresses and physical flash addresses [14]. The FTL chooses where to place each incoming logical sector during a write. If the candidate physical block is occupied with other data, it must be moved and the block must be erased. The FTL then writes the new data and adjusts the map to reflect the position of the new data. While sequential write patterns are easy to handle, non-sequential write patterns can be problematical for the FTL by requiring data copying in order to free up space for each incoming write. In the absolute worst case of continuous 512 byte writes to random addresses, it may be necessary to move a full MLC flash block (512 KB) less 512 bytes for each incoming write, reducing write-lifetime by a factor of 1000. The effect is usually known as *write-amplification* [10] to which we must also add the cost of maintaining even wear across all blocks. Although the worst-case workload is not likely, and the FTL can lessen the negative impact of a non-sequential write workload by maintaining a pool of reserve blocks not included in the drive's advertised capacity, non-sequential workloads will always trigger more erasures than sequential ones.

It is not straightforward to map between reduced write workload and increased write-lifetime. Halving the number of writes will at least double the lifetime. However, the effect can be greater to the extent it also reduces write-amplification. Overwrites are non-sequential by nature. So if overwrites can be eliminated, or out-of-order writes made sequential, there will be both fewer writes and less write-amplification. As explored by Agrawal *et al.* [1], FTL firmware can differ wildly in its ability to handle non-sequential writes. A simple FTL that maps logical sector addresses to physical flash at the granularity of a flash block will suffer huge write-amplification from a non-sequential workload, and therefore will benefit greatly from fewer of such writes. The effect will be more subtle for an advanced FTL that does the mapping at a finer granularity. However, improved sequentiality will reduce internal fragmentation within flash blocks, and therefore will both improve wear-leveling performance and reduce write-amplification.

Write-lifetime depends on the performance of wear-leveling and the write-amplification for a given workload, both of which cannot be measured. However, we can obtain a rough estimate of write-amplification by observing the performance difference between a given workload and a purely sequential one; the degree of observed slowdown should give us some idea of the effective write-amplification. The product manual for the Intel X25-M MLC SSD [13] indicates that this SSD suffers at least a factor of 6 reduction in performance when a random-write workload is compared to a sequential one (sequential write bandwidth of 70 MB/s versus 3.3 K IOPS for random 4 KB writes). Thus, after wear-leveling and other factors are considered, it becomes plausible that practical write-lifetimes, even for advanced FTLs, can be an order of magnitude worse than the optimum.

3 Overview of Griffin

Griffin's design is very simple: it uses a hard disk as a persistent write cache for an MLC-based SSD. All writes are appended to a log stored on the HDD and eventually migrated to the SSD, preferably before subsequent reads. Structuring the write cache as a log allows Grif-

fin to operate the HDD at its fast sequential write mode. In addition to coalescing overwrites, the write cache also increases the sequentiality of the workload observed by the SSD; as described in the previous section, this results in increased write-lifetime.

Since cost is the single biggest barrier to SSD deployment [22], we focus on write caching for cheaper MLC-based SSDs, for which low write-lifetime is a significant constraint. MLC devices are excellent candidates for HDD-based write caching since their sequential write bandwidth is typically equal to that of commodity HDDs, at 70-80 MB/s [13].

Griffin increases the write-lifetime of an MLC-based SSD without increasing total cost significantly; as of this writing, the cost of a 350 GB SATA HDD is around 50 USD, whereas an 128 GB MLC-based SSD is around 300 USD. In comparison, a 128 GB SLC-based SSD, which offers higher write-lifetime than the MLC variant currently costs around 4 to 5 times as much.

Griffin also increases write-lifetime without substantially altering the reliability characteristics of the MLC device. While the HDD write cache represents an additional point of failure, any such event leaves the file system intact on the SSD and only results in the loss of recent data. We discuss failure handling in Section 5.3.

3.1 Other Hybrid Designs

Other hybrid designs using various combinations of RAM, non-volatile RAM, and rotating media are clearly possible. Since a thorough comparative analysis of all the options is beyond the scope of this paper, we briefly describe a few other designs and compare them qualitatively with Griffin.

• **NVRAM as read cache for HDD storage**: Given its excellent random read performance, NVRAM (*e.g.*, an SSD) can work well as a read cache in front of a larger HDD [17, 19, 24]. However, a smaller NVRAM is likely to provide only incremental performance benefits as compared to an OS-based file cache in RAM, whereas a larger NVRAM cache is both costly and subject to wear as the cache contents change. Any design that uses rotating media for primary storage will scale-up in capacity with less cost than Griffin. However, this cost difference is likely to decline as flash memory densities increase.

• **NVRAM as write cache for SSD storage**: The Griffin design can accommodate NVRAM as a write cache in lieu of HDD. The effectiveness of using NVRAM depends on two factors: 1) whether SLC or MLC flash is used; and, 2) the ratio of reads that hit the write cache and thus disrupt sequential logging there. The use of NVRAM can also lead to better power savings. However, all these benefits come at a higher cost than Griffin configured with a HDD cache, especially if SLC flash is used for write caching. Later, we evaluate the Griffin's performance with both SLC and MLC write caches (Section 6.4) and explore the minimum write cache size required (Section 7).

• **RAM as write cache for SSD storage**: RAM can make for a fast and effective write cache, however the overriding problem with RAM is that it is not persistent (absent some power-continuity arrangements). Increasing the RAM size or the timer interval for periodic flushes may reduce the number of writes to storage but only at the cost of a larger window of vulnerability during which a power failure or crash could result in lost updates. Moreover, a RAM-based write cache may not be effective for all workloads; for example, we later show that for certain workloads (Section 6.1.2), over 1 hour of caching is required to derive better write savings; volatile caching is not suitable for such long durations.

3.2 Understanding Griffin Performance

The key challenge faced by Griffin is to increase the write-lifetime of the SSD while retaining its performance on reads. Write caching is a well-known technique for buffering repeated writes to a set of blocks. However, Griffin departs significantly from conventional caching designs, which typically use small, fast, and expensive media (such as volatile RAM or non-volatile battery-backed RAM) to cache writes against larger and slower backing stores. Griffin's HDD write cache is both inexpensive and persistent and can in fact be *larger* than the backing SSD; accordingly, the flushing of dirty data from the write cache to the SSD is not driven by either capacity constraints or synchronous writes.

However, Griffin's HDD write cache is also *slower* than the backing SSD for read operations, which translate into high latency random I/Os on the HDD's log. In addition, reads can disrupt the sequential stream of writes received by the HDD, reducing its logging bandwidth by an order of magnitude. As a result, dirty data has to be flushed to the SSD before it is read again, in order to avoid expensive reads from the HDD.

Griffin's performance is thus determined by competing imperatives — data must be held in the HDD to buffer overwrites, and data must be flushed from the HDD to prevent expensive reads. We quantify these with the following two metrics:

• **Write Savings**: This is the percentage of total writes that is prevented from reaching the SSD. For example, if the hybrid device receives 60M writes and the SSD receives 45M of them, the write savings is 25%. Ideally, we want the write savings to be as high as possible.

• **Read Penalty**: This is the percentage of total reads serviced by the HDD write cache. For example, if the hybrid device receives 50M reads and the HDD receives

1M of these reads, the read penalty is 2%. Ideally, we want the read penalty to be as low as possible.

There will be no read penalty if an oracle informs Griffin in advance of data to be read; all such blocks can be flushed to the SSD before an impending read. With no read penalty, the maximum write savings possible is workload-dependent and is essentially a measure of the frequency of consecutive overwrites without intervening reads. In the worst case, there will be no write savings if there are no overwrites, *i.e.*, no block is ever written consecutively without an intervening read. An idealized HDD write cache achieves the maximum write savings with no read penalty for any workload.

To understand the performance of an idealized HDD write cache, consider the following sequence of writes and reads to a particular block: $WWWRWW$. Without a write cache, this sequence results in one read and five writes to the SSD. An idealized HDD write cache would coalesce consecutive writes and flush data to the SSD immediately before each read, resulting in a sequence of operations to the SSD that contains two writes and one read: WRW. Accordingly, the maximum write savings in this simple example is $3/5$ or 60%.

Griffin attempts to achieve the performance of an idealized HDD write cache by controlling policy along two dimensions: *what data to cache*, and *how long to cache it for*. The choice of policy in each case is informed by the characteristics of real workloads, which we will examine in the next section. Using these different policies, Griffin is able to achieve different points in the trade-off curve between read penalty and write savings.

4 Trace Analysis

In this section, we explore the benefits of HDD-based write caching by analyzing traces from desktop and server environments. Our analysis has two aspects. First, we show that an idealized HDD-based write cache can provide significant write savings for these traces; in other words, overwrites commonly occur in real-world workloads. Second, we look for spatial and temporal patterns in these overwrites that can help determine Griffin's caching policies.

4.1 Description of Traces

Our desktop I/O traces are collected from desktops and laptops running Windows Vista, which were instrumented using the Windows Performance Analyzer. Although we analyzed several desktop traces, we limit our presentation to 12 traces from three desktops due to space limitations.

Most of our server traces are from a previous study by Narayanan *et al.* [21]. These traces were collected from 36 different volumes from 13 servers running Windows Server 2003 SP2. Out of 36 different traces, we used only the most write-heavy data volume traces that have at least one write for every two reads, and have more than 100,000 writes in total (read-intensive workloads already work well on SSDs and do not require write caching). In addition, we also used a Microsoft Exchange server trace, which was collected from a RAID controller managing a terabyte of data.

Trace	Time (hr)	Number of 4 KB I/Os	Read (%)	Write (%)	Max Write Savings (%)	Overwrites in top 1% (%)	Reads in top 1%
D-1A	114	14 M	43	57	46	87	4
D-1B	70	29 M	45	55	39	87	2
D-1C	153	36 M	50	50	52	88	2
D-1D	27	07 M	40	60	64	84	1
D-2A	99	39 M	49	51	39	71	3
D-2B	105	30 M	48	52	36	63	2
D-2C	149	17 M	44	56	58	52	2
D-2D	103	22 M	56	44	52	47	1
D-3A	52	13 M	56	44	43	68	2
D-3B	105	33 M	50	50	56	72	4
D-3C	96	37 M	52	48	47	77	6
D-3D	55	16 M	51	49	51	78	4
S-EXCH	0.25	209 K	59	41	42	34	0
S-PRXY1	167	543 M	65	35	57	99	63
S-SRC10	168	408 M	47	53	14	11	2
S-SRC22	176	16 M	37	63	47	8	2
S-STG1	168	23 M	93	7	93	41	0
S-WDEV2	166	369 K	1	99	94	10	0

Table 1: **Windows Traces.**

Table 1 lists the traces we used for the analysis, where the desktop traces are prefixed by a "D" and server traces by an "S". D-1, D-2, and D-3 represent the three desktops that were traced. EXCH, PRXY1, SRC10/22, STG1, and WDEV2 correspond to traces from a Microsoft Exchange server, firewall or web proxy, source control, web staging, and a test web server. For each trace, the columns 2-5 show the total tracing time, number of I/Os, and read-write percentage.

All the traces contain block-level reads and writes below the NTFS file system cache. Each I/O event specifies the time stamp (in ms), disk number, logical sector number, number of sectors transferred, and type of I/O. Even though the desktop traces contain file system level information such as which file or directory a block access belongs to, the server traces do not have them.

4.2 Ideal Write Savings

Our first objective in the trace analysis is to answer the following question: *do desktop and server I/O traffic have enough overwrites to coalesce and if so, what are the maximum write savings provided by an idealized*

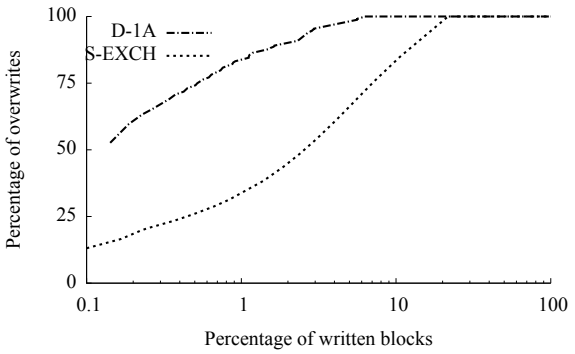


Figure 1: **Distribution of Block Overwrites.**

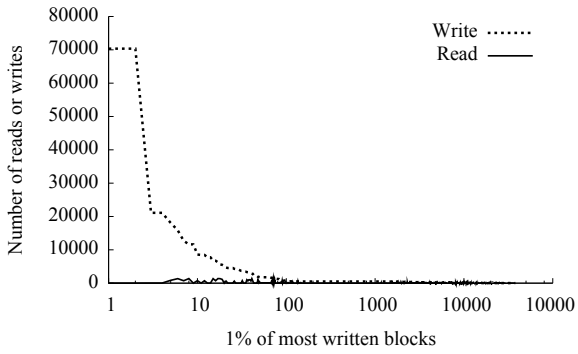


Figure 2: **Reads in Write-Heavy Blocks.**

Rank	Filenames
1	C:\Outlook.ost
2	C:\...\Search\...\Windows.edb
3	C:\$Bitmap
4	C:\Windows\Prefetch\Layout.ini
5	C:\Users\<name>\NTUSER.DAT
6	C:\$Mft

Table 2: **Top Overwritten Files in Desktops.**

HDD write cache? The 6th (highlighted) column in the Table 1 shows the maximum write savings achieved by an idealized write cache that incurs no read penalty.

From the 6th column of Table 1, we observe that an idealized HDD write cache can cut down writes to the SSD significantly. For example, for desktop traces, the maximum write saving is at least 36% (for D-2B) and as much as 64% (for D-1D). The server workloads exhibit similar savings; ideal write savings vary from 14% (S-SRC10) to 94% (S-WDEV2). On an average, desktop and server traces offer write savings of 48.58% and 57.83% respectively. Based on this analysis, the first observation we make is: *desktop and server workloads contain a high degree of overwrites*, and an idealized HDD write cache with no read penalty can achieve significant write savings on them.

Given that an idealized HDD-based write cache has high potential benefits, our next step is to explore the two important policy issues in designing a practical write cache: what do we cache, and how long do we cache it? We investigate these questions in the following sections.

4.3 Spatial Access Patterns

If block overwrites exhibit spatial locality, we can achieve high write savings while caching fewer blocks, reducing the possibility of reads to the HDD. Specifically, we want to find out if some blocks are overwritten more frequently than others. To answer this question, we studied the traces further and make two more observations. First, *there is a high degree of spatial locality in block overwrites*; for example, on an average 1% of the most written blocks contribute to 73% and 34% of the total overwrites in desktop and server traces.

Figure 1 shows the spatial distribution of overwrites for two sample traces: D-1A and S-EXCH. On y-axis, we plot the cumulative distribution of overwrites and in x-axis, we plot the percentage of blocks written. We can notice that a small fraction of the blocks (*e.g.*, 1%) contribute to a large percentage of overwrites (over 70% in D-1A and 33.5% in S-EXCH). For all the traces, we present the percentage of total overwrites that occur in the top 1% of the most overwritten blocks in 7th column of Table 1. We can notice that a small number of blocks absorb most of the overwrite traffic.

The second observation we make is that *the blocks that are most heavily written receive very few reads*. Figure 2 presents the total number of writes and reads in the most heavily written blocks from trace D-1A. We collected the top 1% of the most written blocks and plotted a histogram of the number of writes and reads issued to those blocks. For all the traces, the percentage of total reads that occur in the write-heavy blocks is presented in the last column of Table 1. On average, the top 1% of the blocks in the desktop traces receive 70% of overwrites but only 2.7% of all reads; for the server traces, they receive 0-2% of the reads, excepting S-PRXY1.

To gain some insight into the file-level I/O patterns that cause spatial clustering of overwrites, we compiled a list of the most overwritten files for desktops and present it in Table 2. Not surprisingly, files such as mail boxes, search indexes, registry files, and file system metadata receive most of the overwrites. Some of these files are small enough to fit in the cache (*e.g.*, bitmap or registry entries) and therefore, incur very few reads. We do not report on the most overwritten files in the server traces because they did not contain file-level information. We believe that a similar pattern will be present in other operating systems where majority of overwrites are issued to application-level metadata (*e.g.*, search indexes) and

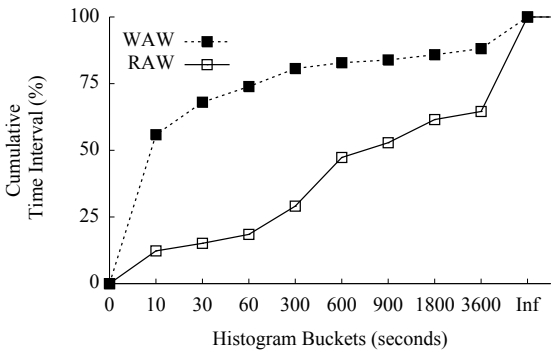


Figure 3: **WAW and RAW Time Intervals.**

system-level metadata (*e.g.*, bitmaps).

At a first glance, such a dense spatial locality of overwritten blocks appears as an opportunity for various optimizations. First, it might suggest that a small cache of few tens of megabytes can be used to handle only the most frequently overwritten blocks. However, separating blocks in this fashion can break the semantic associations of logical blocks (for example, within a file) and make recovery difficult (Section 5.3). Second, a Griffin implementation at the file system-level (Section 7) can easily relocate heavily overwritten files to the HDD. However, when Griffin is implemented as a block device, which is much more tractable in practice, it becomes quite difficult to make use of overwrite-locality lacking file system-level and application-level knowledge.

4.4 Temporal Access Patterns

As mentioned earlier, it is also important to find out how long we can cache a block in the HDD log without incurring expensive reads. To answer this question, we must first understand the temporal access patterns of I/O traces and for that purpose, we define two useful metrics.

Write-After-Write (WAW): WAW is the time interval between two consecutive writes to a block before an intervening read to the same block.

Read-After-Write (RAW): RAW is the time interval between a write and a subsequent read to the same block.

Figure 3 presents the cumulative distribution of the WAW time intervals (indicated by black squares) and the RAW time intervals (indicated by white squares) from 10 seconds to 1 hour for D-1A. Interval larger than 1 hour is indicated by "Inf" on the x-axis. Table 3 presents the WAW and RAW distribution for all the traces.

From Figure 3 and Table 3, we notice that a large percentage of the WAW intervals on desktops are relatively small. In other words, most of the consecutive writes to the same block occur within a short period of time; for example, on average 54% of the total overwrites occur

	WAW				RAW			
Trace	**30 s (%)**	**60 s (%)**	**900 s (%)**	**3600 s (%)**	**30 s (%)**	**60 s (%)**	**900 s (%)**	**3600 s (%)**
D-1A	68	74	84	88	15	18	53	65
D-1B	71	76	87	90	12	16	49	64
D-1C	69	73	81	86	8	9	19	30
D-1D	76	80	89	93	17	18	27	37
D-2A	51	55	69	75	4	6	22	58
D-2B	38	44	62	70	7	8	13	25
D-2C	28	34	59	68	9	9	16	21
D-2D	25	30	56	66	6	7	16	31
D-3A	40	53	71	78	20	22	31	39
D-3B	57	63	71	75	8	10	27	35
D-3C	60	66	73	77	7	8	40	48
D-3D	62	68	75	79	9	16	50	58
S-EXCH	46	54	100	100	9	16	50	58
S-PRXY1	52	64	98	98	12	37	100	100
S-SRC10	2	2	9	10	0	0	4	6
S-SRC22	15	16	17	85	3	3	14	14
S-STG1	6	7	27	41	1	1	9	9
S-WDEV2	7	20	23	23	0	0	0	0

Table 3: **WAW/RAW Distribution**

within the first 30 seconds of the previous write. However, this trend is not so clear in servers, where we see widely varying behaviors, most likely depending upon the specific server workloads. But, we still see benefits from long-term caching: on average, 60% of the overwrites in the server traces occur within an hour of a previous write.

In addition, we also notice that the time between a write to a block and a subsequent read to the same block (*i.e.*, RAW) is relatively long. For example, only an average of 30% the written data is read within 900 seconds of a block write. As with the WAW results, the RAW distribution for the server traces also varies depending on the specific workload.

We believe that the time interval from a write to a subsequent read is large due to large OS-level buffer caches and a smaller percentage of most overwritten blocks; as a result, the buffer cache can service most reads that occur soon after a write, exposing only later reads that are issued after the block evict to the block device. These results are similar to the WAW and RAW results presented in earlier work by Hsu *et al.* [9].

We calculated the WAW and RAW time intervals for the most overwritten files from Table 2. Even though the WAW distribution was similar to the overall traces, RAW time intervals were longer. For example, for the frequently overwritten files, only an average of 21% of the written data is read within 900 seconds of a write.

From this temporal analysis, we make two observations that are important in determining the duration of caching in HDD: first, *intervals between writes and subsequent overwrites are typically short for desktops*; sec-

Trace	Time (hr)	Number of 4 KB I/Os	Read (%)	Write (%)	Max Write Savings (%)	Overwrites in top 1% (%)	Reads in top 1% (%)
D-DEV	164	4 M	27	73	62	72	0
S-SVN	165	241 K	32	68	81	50	0
S-WEB	5	7 M	91	9	81	21	0

Table 4: **Linux Traces.**

	WAW				RAW			
Trace	30 s (%)	60 s (%)	900 s (%)	3600 s (%)	30 s (%)	60 s (%)	900 s (%)	3600 s (%)
D-DEV	9	24	35	45	6	24	84	85
S-SVN	23	32	53	67	2	2	6	10
S-WEB	5	22	46	100	5	9	54	95

Table 5: **Linux WAW/RAW Distribution**

ond, *the time interval between a block write and its consecutive read is large (tens of minutes).*

These observations provide us with insight on how long to cache blocks in the HDD before migrating them to the SSD: long enough to capture a substantial number of overwrites (*i.e.*, higher than some fraction of WAW intervals) but not long enough to receive a substantial number of reads to the HDD (*i.e.*, lower than some fraction of RAW intervals). Using different values for the migration interval clearly allows Griffin to trade-off write savings against read penalty.

4.5 Results from Linux

We also examined Linux block-level traces to find out if they exhibit similar behavior. We used traces from previous work by Bhadkamkar *et al.* [3]. Table 4 presents results from 3 traces: D-DEV is a trace from a development environment; S-SVN consists of traces from SVN and Wiki server; and S-WEB contains traces from a web server. We can see certain similarities between the Linux and Windows traces. For example, in the desktop trace, coalescing of overwrites leads to only 38% of the total writes going to the SSD (and thereby resulting in 62% write savings). Also, we can notice spatial locality in overwrites, with no read I/Os in the top 1% of the most written blocks. Table 5 presents the distribution of WAW and RAW time intervals as was presented for the Windows traces. Unlike Windows, only 50% or less of the overwrites happen within 1 hour, which motivates longer caching time periods in the HDD. Although shown here for completeness, we do not use Linux traces for the rest of the analysis.

4.6 Summary

We find that block overwrites occur frequently in real-world desktop and server workloads, validating the central idea behind Griffin. In addition, overwrites exhibit both spatial and temporal locality, providing useful insight into practical caching policies that can maximize write savings without incurring a high read penalty.

5 Prototype Design and Implementation

Thus far, we have discussed HDD-based write caching in abstract terms, with a view to defining policies that indicate what data to cache in the HDD and when to move it to the SSD. The only metrics of concern have been write savings and read penalty.

However, Griffin's choice and implementation of policies are also heavily impacted by other real-world factors. An important consideration is *migration overhead*, both direct (total bytes) and indirect (loss of HDD sequentiality). For example, a migration schedule provided by a hypothetical oracle may be optimal from the standpoint of write savings and read penalty, but might require data to be migrated constantly in small increments, destroying the sequentiality of the HDD's access patterns.

Another major concern is *fault tolerance*; the HDD in Griffin represents an extra point of failure, and certain policies may leave the hybrid system much more unreliable than an unmodified SSD. For example, a migration schedule that pushes data to the SSD while leaving associated file system metadata on the HDD would be very vulnerable to data loss.

Keeping these twin concerns of migration overhead and fault tolerance in mind, Griffin uses two mechanisms to support policies on what data to cache and how long to cache it: *overwrite ratios* and *migration triggers*.

5.1 Overwrite Ratios

Griffin's default policy is *full caching*, where the HDD caches every write that is issued to the logical address space. An alternate policy is *selective caching*, where only the most overwritten blocks are cached in the HDD. In order to implement selective caching, Griffin computes an overwrite ratio for each block, which is the ratio of the number of overwrites to the number of writes that the block receives. If the overwrite ratio of a block exceeds a predefined value (which we call the *overwrite threshold*), it is written to the HDD log. Full caching is enabled simply by setting the overwrite threshold to zero. As the overwrite threshold is increased, only those blocks which have a higher overwrite ratio – as a result of being frequently overwritten – are cached.

Selective caching has the potential to lower read penalty, as Section 4.3 showed, and to reduce the amount of data migrated. However, an obvious downside of selective caching is its high overhead; it requires Griffin to compute and store per-block overwrite ratios. Additionally, as we will shortly discuss, selective caching also complicates recovery from failures.

5.2 Migration Triggers

Griffin's policy on how long to cache data is determined not by per-block time values, which would be prohibitively expensive to maintain and enforce, but by coarse-grained triggers that cause the entire contents of the HDD cache to be flushed to the SSD. Griffin supports three types of triggers:

Timeout Trigger: This trigger fires if a certain time elapses without a migration. The main advantages of this trigger are that it is simple and predictable. It also bounds the recency of data lost due to HDD failure; a timeout value of 5 minutes will ensure that no write older than 5 minutes will be lost. However, since it does not react to the workload, certain workloads can incur high read penalties.

Read-Threshold Trigger: The read-threshold trigger fires when the measured read penalty since the last migration goes beyond a threshold. The advantage of such an approach is that it allows the read penalty, which could be a reason for Griffin's performance hit, to be bounded. If used in isolation, however, the read-penalty trigger can be subject to pathological scenarios; for example, if data is never read from the device, the measured read penalty will stay at zero and the data will never be moved from the HDD to the SSD. This can result in the HDD running out of space, and also leave the system more vulnerable to data loss on the failure of the HDD.

Migration-Size Trigger: The migration-size trigger fires when the total size of migratable data exceeds a certain size. It is useful in bounding the quantity of data lost on HDD failure. On its own, this trigger is inadequate in ensuring low read penalties or constant migration rates.

Used in concert, these triggers can enable complex migration policies that cover all bases: for example, a policy could state that the read penalty should never be more than 5%, and that no more than 100 MB or 5 minutes worth of data should be lost if the HDD fails.

The actual act of migration is very quick and simple; data is simply read sequentially from the HDD log and written to the SSD. Since the log and the actual file system are on different devices, this process does not suffer from the performance drawbacks of cleaning mechanisms in log-structured file systems [26], where shuttling between the log and the file system on the same device can cause random seeks.

5.3 Failure Handling

Since Griffin uses more than one device to store data, failure recovery is more involved than on a single device.

Power Failures. Power failures and OS crashes can leave the storage system state distributed across the HDD log and the SSD. Recovering the state from the HDD log to the primary SSD storage is simple; Griffin leverages well-developed techniques from log-structured and journaling systems [8, 26] for this purpose. On a restart after a crash, Griffin reads the blockmap that stores the log-block to SSD-block mapping and restores the data that were issued before the system crash.

Device Failures. The HDD or SSD can fail irrecoverably. Since SSD is the primary storage, its failure is simply treated as the failure of the entire hybrid storage, even though the recent writes to the log can be recovered from the HDD. HDD failure can result in the loss of writes that are logged to the disk but not yet migrated to the SSD. The magnitude of the loss depends on both the overwrite ratio and the migration triggers used.

In full caching, since every write is cached, the amount of lost data can be high. However, full caching exports a simple failure semantics; that is, every data block that is available from the SSD is older than every missing write from the HDD. This recovery semantics, where the most recent data writes are lost, is simple and well-understood by file systems. In fact, this can happen even in a single device if the data stored on the device's buffer cache is lost due to say, a power failure.

On the other hand, selective caching minimizes the amount of data loss because it writes fewer blocks in the HDD. However, the semantics of the recovered data is more complex and can lead to unexpected errors: that is, some of the data that is present in the SSD might be more recent than the data that is lost from the HDD because of selective caching.

The migration triggers used directly impact the amount of data loss, as explained in the previous subsection. Timeout and migration-size triggers can be used to tightly bound the recency and quantity of lost data.

5.4 Prototype

We implemented a trace-driven simulator and a user-level implementation for evaluating Griffin. The simulator is used to measure the write savings, HDD read penalties, and migration overheads, whereas the user-level implementation is used for obtaining real latency measurements by issuing the I/Os from the trace to an actual HDD/SSD combo using raw device interfaces.

On a write to a block, Griffin redirects the I/O to the tail of the HDD log and records its new location in an internal in-memory map. The recent contents of the in-

memory map are periodically flushed to the HDD for recovery purposes. On a read to the block, Griffin reads the latest copy of the block from the appropriate device.

Whenever the chosen migration trigger fires, the cached data is migrated from the HDD to the SSD. In order to identify the mapping between the log writes and the logical SSD blocks, Griffin reads the blockmap from the HDD (if it is not already present in memory) and reconstructs the mapping. When migrating, Griffin reads the log contents as sequentially as possible, skipping only the older versions of the data blocks, sorts the logged data based on their logical addresses and writes them back to the SSD. As we show later, this migration improves the sequentiality of the data writes to the SSD.

Even though writes are logged sequentially, the HDD may incur rotational latency. Such rotational latencies can be minimized either by using a small buffer (*e.g.*, 128 KB) to cache writes before writing them to the HDD or by using new mechanisms such as range writes [2].

6 Evaluation

6.1 Policy Evaluation

Although we have several caching and migration policies, we must pick those that are not only effective in reducing the SSD writes but also efficient, practical, and high performing. In this section, we analyze all the policies and pick those that will be used for the evaluation of write savings and performance.

6.1.1 Caching Policies

We evaluate the full and selective caching policies by running different traces through the trace-driven simulator, for different overwrite thresholds; a value of zero for the threshold corresponds to full caching. We then measure the write savings and the read penalty. We disable migrations in these experiments, to compare their performance independent of migration policies.

Figure 4a shows the write savings on y-axis for different traces on x-axis. Each stacked bar per trace plots the cumulative write savings for a specific overwrite threshold. From the figure, we notice that using an overwrite threshold can lower write savings, sometimes substantially as in the server traces.

Figure 4b plots the read penalty on y-axis, where each stacked bar per trace plots the percentage of total reads that hit the HDD for the corresponding overwrite threshold. We observe that a high overwrite threshold has the advantage of eliminating a large fraction of HDD reads.

From Figures 4a and 4b, it is apparent that full caching has the advantage of providing the maximum write savings, but suffers from a higher read penalty as well. It is important to note, however, that the read penalty reported in Figure 4b *is an upper bound on the actual read penalty*, since in this experiment data is never migrated from the HDD and all reads to a block that occur after a preceding write must be served from the HDD. In addition, as described in Section 5.1, a non-zero value on the overwrite threshold comes at a high overhead, requiring Griffin to compute and maintain per-block overwrite ratios. It also complicates recovery from failures.

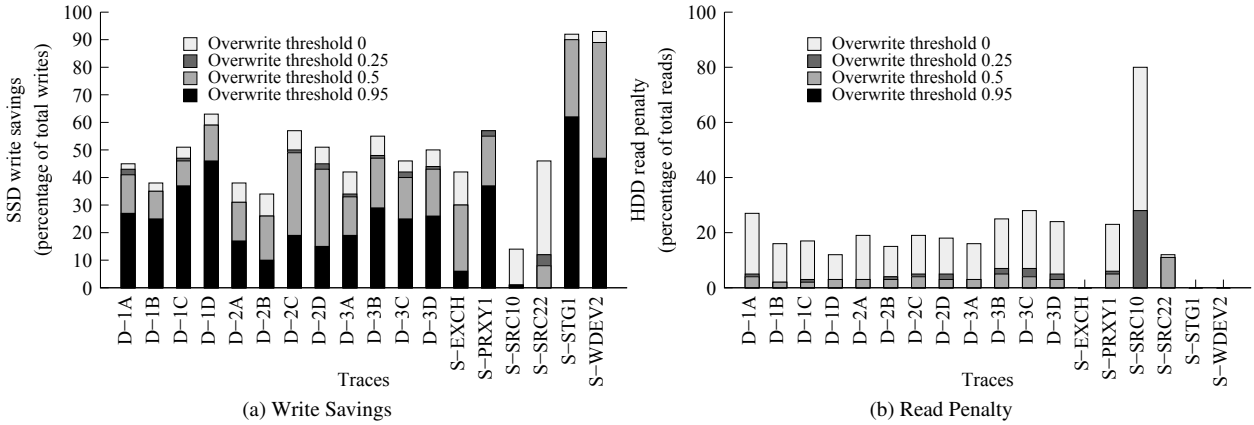
These factors lead us to the conclusion that full caching wins in most cases and therefore, in the remaining experiments, we use full caching exclusively.

6.1.2 Migration Policies

Next, we evaluate different migration policies using the trace-driven simulator. In addition to the write savings, we also measure the inter-migration interval, read penalty, and migration sizes. We start by plotting the write savings for timeout triggers in Figure 5a. We observe that logging for 15 minutes (900 s) gives most of the write savings (over 80% in nearly all cases). For some traces, such as S-STG1, over 1 hour of caching is required to derive better write savings. The durability and large size of the HDD cache allows us to meet such long caching requirements; alternative mechanisms such as volatile in-SSD caches are not large enough to hold writes for more than 10s of seconds.

We also show the read penalty for different timeout values in Figure 5b. We find that the read penalty is low (less than 20%) in most cases except one (S-PRXY1). In particular, read penalty is much lower than the no-migration upper bound reported in Figure 4b, underlining the fact that full caching is not hampered by high read penalties because of frequent migrations. In addition, we also find that timeout-based migration bounds the migration size. The average migration size varied between 91 MB to 344 MB for timeout values of 900 to 3600 seconds.

Figure 6a shows the write savings for read-threshold triggers. Even a tight read-threshold bound of 1% produces write savings similar to those for timeout triggers for most traces. However, the drawback of a smaller read-threshold is frequent migration. Figure 6b plots the average time between two consecutive migrations as a log scale on y-axis for various traces and read penalties. We observe that for most traces, a smaller read-threshold triggers more frequent migrations, separated by as low as 6 seconds as in S-PRXY1. Interestingly, for some traces such as S-WDEV2, which has a very small percentage of reads, even a small read-trigger such as 1% never fires and therefore, the data remains on HDD cache for a long time. As explained earlier (Section 5.3), such behavior increases the magnitude of data loss on HDD failure. The



(a) Write Savings

(b) Read Penalty

Figure 4: **Write Savings and Read Penalty Under Full and Selective Caching.**

migration size varied widely from an average of 129 MB to 1823 MB for 1% to 10% read-thresholds.

Since timeout-based migration was also bounding the migration size, we simplified our composite trigger to consist of a timeout-based trigger combined with a read-threshold trigger. For the rest of the analysis, we use full caching with the composite migration trigger.

6.2 Increased Sequentiality

One of the additional benefits of aggressive write caching is that as writes get accumulated for random blocks, the sequentiality of writes to the SSD increases. Such increased sequentiality in write traffic is an important factor in improving the performance and lifetime of SSDs as it reduces write amplification [10].

Figure 7 plots the percentage of sequential page writes sent to the SSD with and without Griffin, on the desktop and server traces. We use the trace-driven simulator to obtain these results. We count a page write as sequential if the preceding write occurs to an adjacent page. For most traces, Griffin substantially increases the sequentiality of writes observed by the SSD.

6.3 Lifetime Improvement

As mentioned in Section 2, it is not straightforward to compute the exact lifetime improvement from write savings as it depends heavily on the workload and flash firmware. However, given the write I/O accesses, we can find the lower bound and upper bound of the flash block erasures, assuming a perfectly optimal and an extremely simple FTL, respectively.

We ran all the traces on our simulator with full caching and composite migration trigger. The I/O writes are fed into two FTL models to calculate the erasure savings. Ideal FTL assumes a page-level mapping and issues all writes sequentially, incurring fewer erasures. Therefore, erasure savings are smaller on ideal FTL because it is already good at reducing erasures. Simple FTL uses a coarse-grained block-level mapping, where if a write is issued to a physical page that cannot be overwritten, then the block is erased. Based on these models, Figure 8 presents the SSD block-erasure savings, which can directly translate into lifetime improvement.

6.4 Latency Measurements

Finally, we measure Griffin's performance on real HDDs and SSDs using our user-level implementation. We use four different configurations for Griffin's write cache: a slow HDD, a fast HDD, a slow SSD, and a fast SSD. In all the measurements, an MLC-based SSD was used as the primary store. We used the following devices: a Barracuda 7200 RPM HDD, a Western Digital 10K RPM HDD, an Intel X25-M 80 GB SSD with MLC flash, and an Intel X25-E 32 GB SSD with SLC flash with a sequential write throughput of 80 MB/s, 118 MB/s, 70 MB/s, and 170 MB/s respectively. When MLC-based SSD is used for write caching, we used Intel X25-M SSDs as the write cache as well as the primary storage.

Since each trace is several days long, we picked only 2 hours of I/Os that stress the Griffin framework. Specifically, we selected two 2-hour segments, T_1 and T_2, out of all the desktop traces that have a large number of total reads and writes per second that hit the cache. T_2 also happened to contain the most number of I/Os in a 2 hour segment. These two trace segments represent I/O streams that stress Griffin to a large extent. We ran each of these trace segments under full caching with a migration timeout of 900 seconds; Griffin's in-memory blockmap was flushed every 30 seconds. The average migration sizes are 2016 MB and 2728 MB for T_1 and T_2.

Figure 9 compares the latencies (relative to the de-

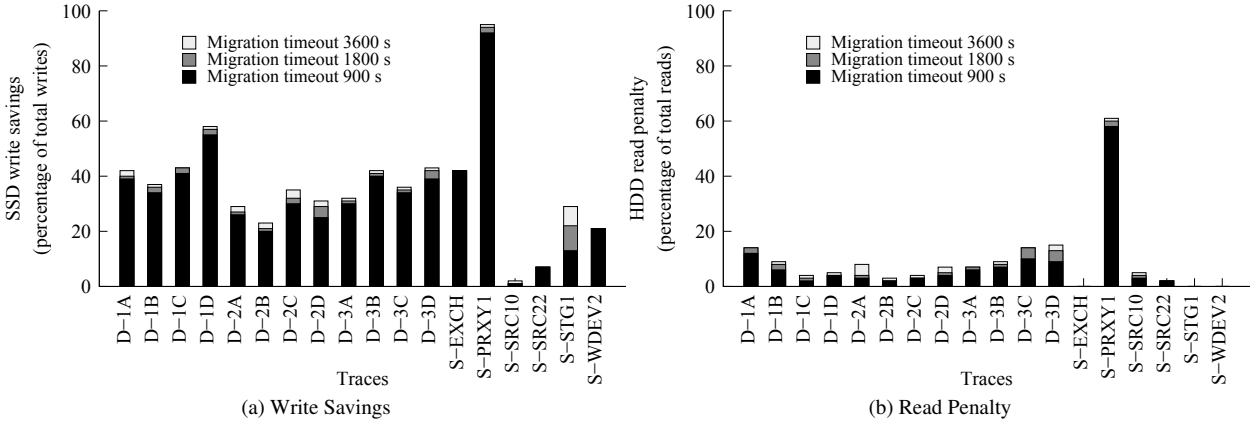


Figure 5: **Write Savings and Read Penalty in Timeout-based Migration.**

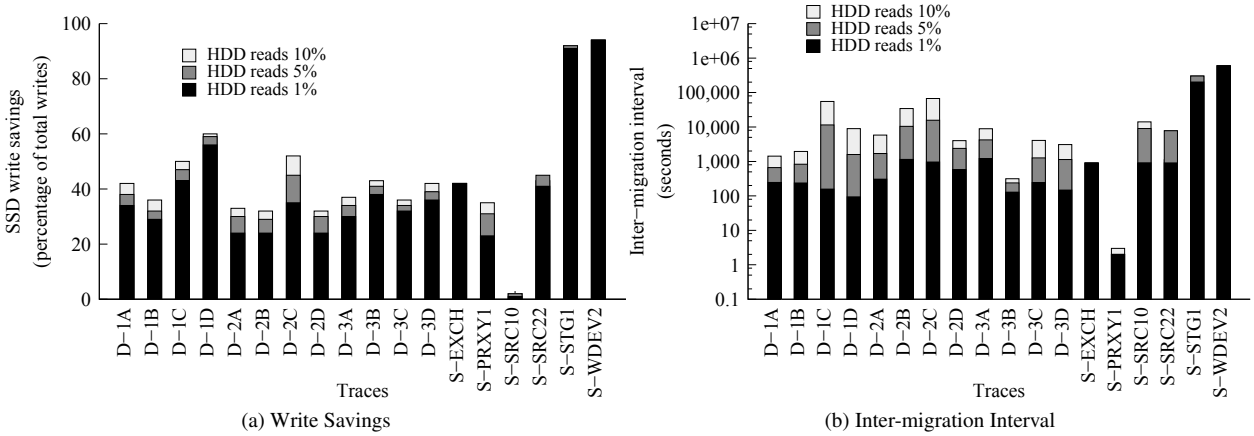


Figure 6: **Write Savings and Inter-migration Interval in Reads-Threshold Migration.**

fault MLC-based SSD) of all I/Os, reads, and writes with different write caches. Unsurprisingly, Griffin performs better than the default SSD in all the configurations (with HDDs or SSDs as its write cache). This is because of two reasons: first, write performance improves because of the excellent sequential throughput of the write caches (HDD or SSD); second, read latency also improves because of the reduced write load on the primary SSD. For example, even when using a slower 7200 RPM HDD as a cache, Griffin's average relative I/O latency is 0.44. That is, Griffin reduces the I/O latencies by 56%. Overall performance of Griffin when using an MLC-based or SLC-based SSD as the write cache is better than the HDD-based write cache because of the better read latencies of SSD. While it is not a fair comparison, this performance analysis brings the high-level point that even when a HDD, which is slower than an SSD for most cases, is introduced in the storage hierarchy the performance of the overall system does not degrade. Figure 9 also shows that using another SSD as a write cache instead of an HDD gives faster performance. But, this comes at a much higher cost because of the price differences between an HDD and SSD. Given the excellent performance of Griffin even with a single HDD, we may explore setups where a single HDD is used as a cache for multiple SSDs (Section 7).

7 Discussion

• **File system-based designs:** Griffin could have been implemented at the file system level instead of the block device level. There are three potential advantages of such an approach. First, a file system can leverage knowledge of the semantic relationships between blocks to better exploit the spatial locality described in Section 4.3. Second, it is possible that Griffin can be easily implemented by modifying existing journaling file systems to store the update journal on the HDD and the actual data on the SSD, though current journaling file systems are

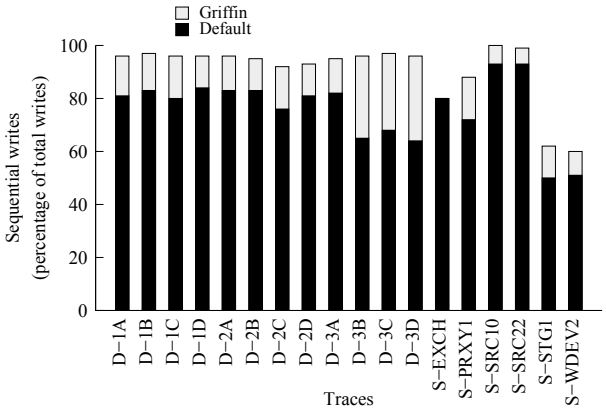


Figure 7: **Improved Sequentiality.**

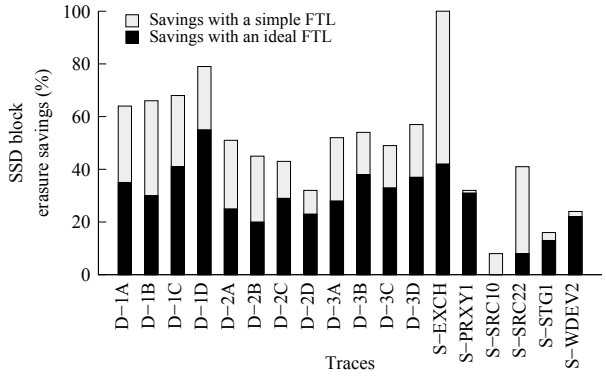


Figure 8: **Improved Lifetime.**

typically designed to store only metadata updates in the journal and many of the overwrites we want to buffer occur within user data.

The third advantage of a file system design is its access to better information, which can enable it to approach the performance of an idealized HDD write cache. Recall that the idealized cache requires an oracle that notifies it of impending reads to blocks just before they occur, so dirty data can be migrated in time to avoid reads from the HDD. At the block level, such an oracle does not exist and we had to resort to heuristic-based migration policies. However, at the file system level, evictions of blocks from the buffer cache can be used to signal impending reads. As long as the file system stores a block in its buffer cache, it will not issue reads for that block to the storage device; once it evicts the block, any subsequent read has to be serviced from the device. Accordingly, a policy of migrating blocks from the HDD to the SSD upon eviction from the buffer cache will result in the maximum write savings with no read penalty.

However, a block device has the significant advantage of requiring no modification to the software stack, working with any OS or architecture. Additionally, our evaluation showed that the simple device-level migration policies we use are very effective in approximating the performance of an idealized cache.

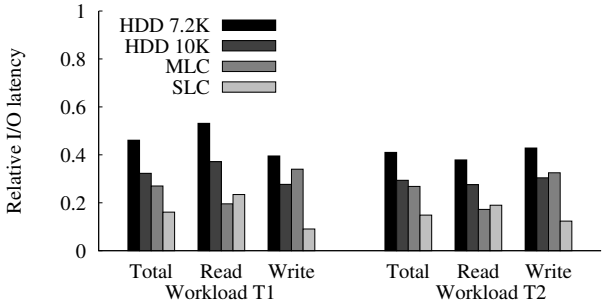


Figure 9: **Relative I/O Latencies for Different Write Caches.**

• **Flash as write cache:** While Griffin uses an HDD as a write cache, it could alternatively have used a small SSD and achieved better performance (Section 6.4). Since SLC flash is expensive, it is crucial that the size of the write cache be small. However, the write cache must also sustain at least as many erasures as the backing MLC-based SSD, requiring a certain minimum size.

Since each SLC block can endure 10 times the erasures of an MLC block, an SLC device subjected to the same number of writes as the MLC device would need to be a tenth as large as the MLC to last as long. If the SLC receives twice as many writes as the MLC, it would need to be a fifth as large.

Consequently, a caching setup that achieves a write savings of 50% – and as a result, sends twice as many writes to the SLC than the MLC – requires an SLC cache that's at least a fifth of the MLC. For example, if the MLC device is 80 GB, then we need an SLC cache of at least 16 GB. In this analysis we assumed an ideal FTL that performs page-level mapping, a perfectly sequential write stream, and identical block sizes for MLC and SLC devices. If the MLC's block size is twice as large as the SLC's block size, as is the case for current devices, the required SLC size stays at a fifth for a perfectly sequential workload, but will drop for more random workloads; we omit the details of the block size analysis for brevity. We believe that a 16 GB SLC write cache (for an 80 GB MLC primary store) will continue to be expensive enough to justify Griffin's choice of caching medium.

• **Power consumption:** One of the main concerns that might arise in the design of Griffin is its power consumption. Since HDDs consume more power than SSDs, Griffin's power budget is higher than that of a regular SSD. One way to mitigate this problem is to use a smaller, more power-efficient HDD such as an 1.8 inch drive that offers marginally lower bandwidth; for example, Toshiba's 1.8 inch HDD [28] consumes about 1.1 watts to seek and about 1.0 watts to read or write, which

is comparable to the power consumption of Micron SSD [18], thereby offering a tradeoff between power, performance, and lifetime. Additionally, desktop workloads are likely to have intervals of idle time during which the HDD cache can be spun down to save power.

Finally, we can potentially use a single HDD as a write cache for multiple SSDs, reducing the power premium per SSD (as well as the hardware cost). Going by the Intel X25-M's specifications, a single SSD supports 3.3K random write IOPS, or around 13 MB/s, whereas a HDD can support 70 to 80 MB/s of sequential writes. Accordingly, a single HDD can keep up with multiple SSDs if they are all operating on completely random workloads, though non-trivial engineering is required for disabling caching whenever the data rate of the combined workloads exceeds HDD speed.

8 Related Work

SSD Lifetimes: SSD lifetimes have been evaluated in several previous studies [6, 7, 20]. The consensus from these studies is that both the reliability and performance of the MLC-based SSDs degrade over time. For example, the bit error rates increase sharply and the erase times increase (by as much as three times) as SSDs reach the end of their lifetime. These trends motivate the primary goal of our work, which is to reduce the number of SSD erasures, thus increasing its lifetime. With less wear, an SSD can provide a higher performance as well.

Disk + SSD: Various hybrid storage devices have been proposed in order to combine the positive properties of rotating and solid state media. Most previous work employs the SSD as a cache on top of the hard disk to improve read performance. For example, Intel's Turbo Memory [17] uses NAND-based non-volatile memory as an HDD cache. Operating system technologies such as Windows ReadyBoost [19] use flash memory, for example in the form of USB drives, to cache data that would normally be paged out to an HDD. Windows ReadyDrive [24] works on hybrid ATA drives with integrated flash memory, which allow reads and writes even when the HDD is spun down.

Recently, researchers have considered placing HDDs and SSDs at the same level of the storage hierarchy. For example, Combo Drive [25] is a heterogeneous storage device in which sectors from the SSD and the HDD are concatenated to form a continuous address range, where data is placed based on heuristics. Since the storage address space is divided among two devices, a failure in the HDD can render the entire file system unusable. In contrast, Griffin uses the HDD only as a cache allowing it to expose an usable file system even in the event of an HDD failure (albeit with some lost updates). Similarly, Koltsidas *et al.* have proposed to split a database store between the two media based on a set of on-line algorithms [15]. Sun's Hybrid Storage Pools consist of large clusters of SSDs and HDDs to improve the performance of data access on multi-core systems [4].

In contrast to the above mentioned works, we use the HDD as a write cache to extend SSD lifetime. Although using the SSD as a read cache may offer some benefit in laptop and desktop scenarios, Narayanan *et al.* have demonstrated that their benefit in the enterprise server environment is questionable [22]. Moreover, any system that forces all writes through a relatively small amount of flash memory will wear through the available erase cycles very quickly, greatly diminishing the utility of such a scheme. Setups with the HDD and SSD arranged as siblings may reduce erase cycles and provide low-latency read access, but can incur seek latency on writes if the hard disk is not structured as a log. Additionally, HDD failure can result in data loss since it is a first-class partition and not a cache.

SLC + MLC: Recently, hybrid SSD devices with both SLC and MLC memory have been introduced. For example, Samsung has developed a hybrid memory chip that contains both SLC and MLC flash memory blocks [27]. Alternatively, an MLC flash memory cell can be programmed either as a single-level or multi-level cell; FlexFS utilizes this by partitioning the storage dynamically into SLC and MLC regions according to the application requirements [16].

Other architectures use SLC chips as a log for caching writes to MLC [5, 12]. These studies emphasize the performance gains that the SLC log provides but do not investigate the effect on system lifetime. As we described in Section 7, a small SLC write cache will wear out faster than the MLC device, and larger caches are expensive.

Disk + Disk: Hu *et al.* proposed an architecture called Disk Caching Disk (DCD), where an HDD is used as a log to convert the small random writes into large log appends. During idle times, the cached data is de-staged from the log to the underlying primary disk [11, 23]. While DCD's motivation is to improve performance, our primary goal is to increase the SSD lifetime.

9 Conclusion

As new technologies are born, older technology might take a new role in the process of system evolution. In this paper, we show that hard disk drives, which have been extensively used as a primary store, can be used as a cache for MLC-based SSDs. Griffin's design is motivated by the workload and hardware characteristics. After a careful evaluation of Griffin's policies and performance, we show that Griffin has the potential to improve SSD lifetime significantly without sacrificing performance.

10 Acknowledgments

We are grateful to our shepherd, Jason Nieh, and the anonymous reviewers for their valuable feedback and suggestions. We thank Vijay Sundaram and David Fields from the Windows Performance Team for providing us the Windows desktop traces. We also thank Dushyanth Narayanan from Microsoft Research Cambridge and Prof. Raju Rangaswami from Florida International University for keeping their traces publicly available. Finally, we extend our thanks to Marcos Aguilera, John Davis, Moises Goldszmidt, Butler Lampson, Roy Levin, Dahlia Malkhi, Mike Schroeder, Kunal Talwar, Yinglian Xie, Fang Yu, Lidong Zhou, and Li Zhuang for their insightful comments.

References

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *Proceedings of USENIX Annual Technical Conference*, pages 57–70, 2008.

[2] 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, CA, December 2008.

[3] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis. BORG: Block-reORGanization for Self-optimizing Storage Systems. In *Proceedings of the File and Storage Technologies Conference*, pages 183–196, San Francisco, CA, Feb. 2009.

[4] R. Bitar. Deploying Hybrid Storage Pools With Sun Flash Technology and the Solaris ZFS File System. Technical Report SUN-820-5881-10, Sun Microsystems, October 2008.

[5] L.-P. Chang. Hybrid solid-state disks: Combining heterogeneous NAND flash in large SSDs. In *Proceedings of the 13th Asia South Pacific Design Automation Conference*, pages 428–433, Jan. 2008.

[6] P. Desnoyers. Empirical evaluation of nand flash memory performance. In *First Workshop on Hot Topics in Storage and File Systems (HotStorage'09)*, 2009.

[7] 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 IEEE/ACM International Symposium on Microarchitecture*, pages 24–33, 2009.

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

[9] W. W. Hsu and A. J. Smith. Characteristics of I/O traffic in personal computer and server workloads. *IBM Systems Journal*, 42(2):347–372, 2003.

[10] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka. Write amplification analysis in flash-based solid state drives. In *SYSTOR 2009: The Israeli Experimental Systems Conference*, 2009.

[11] Y. Hu and Q. Yang. Dcd - disk caching disk: A new approach for boosting i/o performance. In *Proceedings of the International Symposium on Computer Architecture*, pages 169–178, 1996.

[12] S. Im and D. Shin. Storage architecture and software support for SLC/MLC combined flash memory. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1664–1669, 2009.

[13] Intel Corporation. Intel X18-M/X25-M SATA Solid State Drive. `http://download.intel.com/design/flash/nand/mainstream/mainstream-sata-ssd-datasheet.pdf`.

[14] H. Kim and S. Ahn. BPLRU: a buffer management scheme for improving random writes in flash storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–14, 2008.

[15] I. Koltsidas and S. Viglas. Flashing up the storage layer. *Proceedings of the VLDB Endowment*, 1(1):514–525, 2008.

[16] S. Lee, K. Ha, K. Zhang, J. Kim, and J. Kim. FlexFS: A Flexible Flash File System for MLC NAND Flash Memory. In *Proceedings of the USENIX Annual Technical Conference*, San Diego, CA, June 2009.

[17] J. Matthews, S. Trika, D. Hensgen, R. Coulson, and K. Grimsrud. Intel®turbo memory: Nonvolatile disk caches in the storage hierarchy of mainstream computer systems. *Transactions on Storage*, 4(2):1–24, 2008.

[18] Micron. C200 1.8-Inch SATA NAND Flash SSD. `http://download.micron.com/pdf/datasheets/realssd/realssd_c200_1_8.pdf`.

[19] Microsoft Corporation. Microsoft Windows ReadyBoost. `http://www.microsoft.com/windows/windows-vista/features/readyboost.aspx`.

[20] N. Mielke, T. Marquart, N. Wu, J. Kessenich, H. Belgal, E. Schares, F. Trivedi, E. Goodness, and L. R. Nevill. Bit error rate in NAND Flash memories. In *IEEE International Reliability Physics Symposium (IRPS)*, pages 9–19, April 2008.

[21] D. Narayanan, A. Donnelly, and A. I. T. Rowstron. Write offloading: Practical power management for enterprise storage. In *Proceedings of the File and Storage Technologies Conference*, pages 253–267, San Jose, CA, Feb. 2008.

[22] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron. Migrating server storage to SSDs: analysis of tradeoffs. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 145–158, 2009.

[23] T. Nightingale, Y. Hu, and Q. Yang. The design and implementation of a dcd device driver for unix. In *Proceedings of the USENIX Annual Technical Conference*, pages 295–307, 1999.

[24] Panabaker, Ruston. Hybrid Hard Disk and ReadyDrive Technology: Improving Performance and Power for Windows Vista Mobile PCs . `http://www.microsoft.com/whdc/system/sysperf/accelerator.mspx`.

[25] H. Payer, M. A. Sanvido, Z. Z. Bandic, and C. M. Kirsch. Combo drive: Optimizing cost and performance in a heterogeneous storage device. *First Workshop on Integrating Solid-state Memory into the Storage Hierarchy*, 1(1):1–8, 2009.

[26] 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.

[27] Samsung. Fusion Memory: Flex-OneNAND. `http://www.samsung.com/global/business/semiconductor/products/fusionmemory/Products_FlexOneNAND.html`.

[28] Toshiba. MK1214GAH (HDD1902) 1.8-inch HDD PMR 120GB. `http://sdd.toshiba.com/main.aspx?Path=StorageSolutions/1.8-inchHardDiskDrives/MK1214GAH/MK1214GAHSpecifications`.

Write Endurance in Flash Drives: Measurements and Analysis

Simona Boboila
Northeastern University
360 Huntington Ave.
Boston, MA 02115
simona@ccs.neu.edu

Peter Desnoyers
Northeastern University
360 Huntington Ave.
Boston, MA 02115
pjd@ccs.neu.edu

Abstract

We examine the write endurance of USB flash drives using a range of approaches: chip-level measurements, reverse engineering, timing analysis, whole-device endurance testing, and simulation. The focus of our investigation is not only measured endurance, but underlying factors at the level of chips and algorithms—both typical and ideal—which determine the endurance of a device.

Our chip-level measurements show endurance far in excess of nominal values quoted by manufacturers, by a factor of as much as 100. We reverse engineer specifics of the Flash Translation Layers (FTLs) used by several devices, and find a close correlation between measured whole-device endurance and predictions from reverse-engineered FTL parameters and measured chip endurance values. We present methods based on analysis of operation latency which provide a non-intrusive mechanism for determining FTL parameters. Finally we present Monte Carlo simulation results giving numerical bounds on endurance achievable by any on-line algorithm in the face of arbitrary or malicious access patterns.

1 Introduction

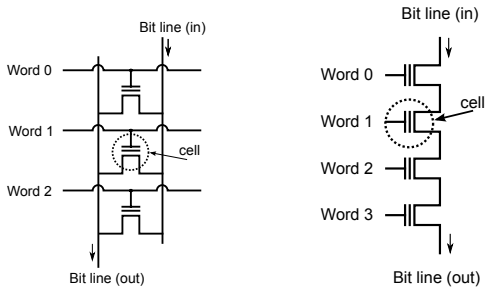
In recent years flash memory has entered widespread use, in embedded media players, photography, portable drives, and solid-state disks (SSDs) for traditional computing storage. Flash has become the first competitor to magnetic disk storage to gain significant commercial acceptance, with estimated shipments of 5×10^{19} bytes in 2009 [10], or more than the amount of disk storage shipped in 2005 [31].

Flash memory differs from disk in many characteristics; however, one which has particular importance for the design of storage systems is its limited *write endurance*. While disk drive reliability is mostly unaffected by usage, bits in a flash chip will fail after a limited number of writes, typical quoted at 10^4 to 10^5 depending on the specific device. When used with applications expecting a disk-like storage interface, e.g. to implement a FAT or other traditional file system, this results in over-use of a small number of blocks and early failure. Almost all flash devices on the market—USB drives, SD drives, SSDs, and a number of others—thus implement internal *wear-leveling* algorithms, which map application block addresses to physical block addresses, and vary this mapping to spread writes uniformly across the device.

The endurance of a flash-based storage system such as a USB drive or SSD is thus a function of both the parameters of the chip itself, and the details of the wear-leveling algorithm (or *Flash Translation Layer*, FTL) used. Since measured endurance data is closely guarded by semiconductor manufacturers, and FTL details are typically proprietary and hidden within the storage device, the broader community has little insight into the endurance characteristics of these systems. Even empirical testing may be of limited utility without insight into which access patterns represent worst-case behavior.

To investigate flash drive endurance, we make use of an array of techniques: chip-level testing, reverse engineering and timing analysis, whole device testing, and analytic approaches. Intrusive tests include chip-level testing—where the flash chip is removed from the drive and tested without any wear-leveling—and reverse engineering of FTL algorithms using logic analyzer probing. Analysis of operation timing and endurance testing conducted on the entire flash drive provides additional information; this is augmented by analysis and simulation providing insight into achievable performance of the wear-leveling algorithms used in conjunction with typical flash devices.

The remainder of the paper is structured as follows. Section 2 presents the basic information about flash memory technology, FTL algorithms, and related work. Section 3 discusses our experimental results, including chip-level testing (Section 3.1), details of reverse-engineered FTLs (3.2), and device-level testing (3.3).



(a) NOR Flash (b) NAND Flash

Figure 1: Flash circuit structure. NAND flash is distinguished by the series connection of cells along the bit line, while NOR flash (and most other memory technologies) arrange cells in parallel between two bit lines.

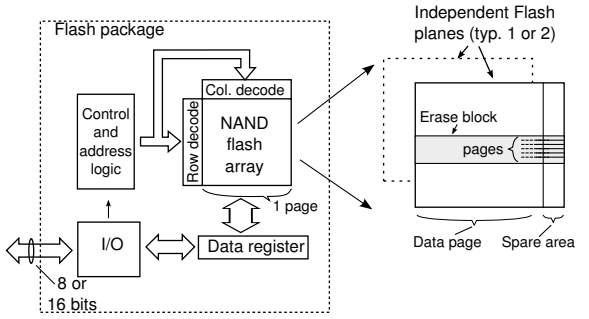


Figure 2: Typical flash device architecture. Read and write are both performed in two steps, consisting of the transfer of data over the external bus to or from the data register, and the internal transfer between the data register and the flash array.

Section 4 presents a theoretical analysis of wear-leveling algorithms, and we conclude in Section 5.

2 Background

NAND flash is a form of electrically erasable programmable read-only memory based on a particularly space-efficient basic cell, optimized for mass storage applications. Unlike most memory technologies, NAND flash is organized in *pages* of typically 2K or 4K bytes which are read and written as a unit. Unlike block-oriented disk drives, however, pages must be erased in units of *erase blocks* comprising multiple pages—typically 32 to 128—before being re-written.

Devices such as USB drives and SSDs implement a re-writable block abstraction, using a Flash Translation Layer to translate logical requests to physical read, program, and erase operations. FTL algorithms aim to maximize endurance and speed, typically a trade-off due to the extra operations needed for wear-leveling. In addition, an FTL must be implementable on the flash controller; while SSDs may contain 32-bit processors and megabytes of RAM, allowing sophisticated algorithms, some of the USB drives analyzed below use 8-bit controllers with as little as 5KB of RAM.

2.1 Physical Characteristics

We first describe in more detail the circuit and electrical aspects of flash technology which are relevant to system software performance; a deeper discussion of these and other issues may be found in the survey by Sanvido *et al* [29]. The basic cell in a NAND flash is a MOSFET transistor with a floating (i.e. oxide-isolated) gate. Charge is tunnelled onto this gate during write operations, and removed (via the same tunnelling mechanism) during erasure. This stored charge causes changes in V_T, the threshold or turn-on voltage of the cell transistor, which may then be sensed by the read circuitry. NAND flash is distinguished from other flash technologies (e.g. NOR flash, E^2PROM) by the tunnelling mechanism (Fowler-Nordheim or FN tunnelling) used for both programming and erasure, and the series cell organization shown in Figure 1(b).

Many of the more problematic characteristics of NAND flash are due to this organization, which eliminates much of the decoding overhead found in other memory technologies. In particular, in NAND flash the only way to access an individual cell for either reading or writing is *through* the other cells in its bit line. This adds noise to the read process, and also requires care during writing to ensure that adjacent cells in the string are not disturbed. (In fact, stray voltage from writing and reading may induce errors in other bits on the string, known as *program disturbs* and *read disturbs*.) During erasure, in contrast, all cells on the same bit string are erased.

Individual NAND cells store an analog voltage; in practice this may be used to store one of two voltage levels (*Single-Level Cell* or SLC technology) or between 4 and 16 voltage levels—encoding 2 to 4 bits—in what is known as Multi-Level Cell (MLC) technology. These cells are typically organized as shown in the block diagram in Figure 2. Cells are arranged in pages, typically containing 2K or 4K bytes plus a spare area of 64 to 256 bytes for system overhead. Between 16 and 128 pages make up an *erase block*, or *block* for short, which are then grouped into a flash *plane*. Devices may contain independent flash planes, allowing simultaneous operations for higher performance. Finally, a static RAM buffer holds data before writing or after reading, and data is transferred to and from this buffer via an 8- or 16-bit wide bus.

2.2 Flash Translation Layer

As described above, NAND flash is typically used with a *flash translation layer* implementing a disk-like interface of addressable, re-writable 512-byte blocks, e.g. over an interface such as SATA or SCSI-over-USB. The FTL maps logical addresses received over this interface (*Logical Page Numbers* or LPNs) to physical addresses in the flash chip (*Physical Page Numbers*, PPNs) and manages the details of erasure, wear-leveling, and garbage collection [2, 3, 17].

Mapping schemes: A flash translation layer could in theory maintain a map with an entry for each 512-byte logical page containing its corresponding location; the overhead of doing so would be high, however, as the map for a 1GB device would then require 2M entries, consuming about 8MB; maps for larger drives would scale proportionally. FTL resource requirements are typically reduced by two methods: *zoning* and larger-granularity mapping.

Zoning refers to the division of the logical address space into regions or *zones*, each of which is assigned its own region of physical pages. In other words, rather than using a single translation layer across the entire device, multiple instances of the FTL are used, one per zone. The map for the current zone is maintained in memory, and when an operation refers to a different zone, the map for that zone must be loaded from the flash device. This approach performs well when there is a high degree of locality in access patterns; however it results in high overhead for random operation. Nonetheless it is widely used in small devices (e.g. USB drives) due to its reduced memory requirements.

By mapping larger units, and in particular entire erase blocks, it is possible to reduce the size of the mapping tables even further [8]. On a typical flash device (64-page erase blocks, 2KB pages) this reduces the map for a 1GB chip to 8K entries, or even fewer if divided into zones. This reduction carries a cost in performance: to modify a single 512-byte logical block, this block-mapped FTL would need to copy an entire 128K block, for an overhead of $256\times$.

Hybrid mapping schemes [19, 20, 21, 25] augment a block map with a small number of reserved blocks (*log* or *update* blocks) which are page mapped. This approach is targeted to usage patterns that exhibit block-level temporal locality: the pages in the same logical block are likely to be updated again in the near future. Therefore, a compact fine-grained mapping policy for log blocks ensures a more efficient space utilization in case of frequent updates.

Garbage collection: Whenever units smaller than an erase block are mapped, there can be *stale* data: data which has been replaced by writes to the same logical address (and stored in a different physical location) but which has not yet been erased. In the general case recovering these pages efficiently is a difficult problem. However in the limited case of hybrid FTLs, this process consists of merging log blocks with blocks containing stale data, and programming the result into one or more free blocks. These operations are of the following types: *switch merges*, *partial merges*, and *full merge* [13].

A *switch merge* occurs during sequential writing; the log block contains a sequence of pages exactly replacing an existing data block, and may replace it without any further operation; the old block may then be erased. A *partial merge* copies valid pages from a data block to the log block, after which the two may be switched. A *full merge* is needed when data in the log block is out of order; valid pages from the log block and the associated data block are copied together into a new free block, after which the old data block and log block are both erased.

Wear-leveling: Many applications concentrate their writes on a small region of storage, such as the file allocation table (FAT) in MSDOS-derived file systems. Naïve mechanisms might map these logical regions to similar-sized regions of physical storage, resulting in premature device failure. To prevent this, *wear-leveling* algorithms are used to ensure that writes are spread across the entire device, regardless of application write behavior; these algorithms [11] are classified as either dynamic or static. *Dynamic wear-leveling* operates only on overwritten blocks, rotating writes between blocks on a free list; thus if there are m blocks on the free list, repeated writes to the same logical address will cause $m + 1$ physical blocks to be repeatedly programmed and erased. *Static wear-leveling* spreads the wear over both static and dynamic memory regions, by periodically swapping active blocks from the free list with static randomly-chosen blocks. This movement incurs additional overhead, but increases overall endurance by spreading wear over the entire device.

2.3 Related Work

There is a large body of existing experimental work examining flash memory performance and endurance; these studies may be broadly classified as either circuit-oriented or system-oriented. Circuit-level studies have examined the effect of program/erase stress on internal electrical characteristics, often using custom-fabricated devices to remove the internal control logic and allow measurements of the effects of single program or erase steps. A representative study is by Lee et al. at Samsung [24], examining both program/erase cycling and hot storage effects across a range of process technologies. Similar studies include those by Park et al. [28] and Yang et al. [32], both also at Samsung. The most recent work

Device	Size (bits)	Cell	Nominal endurance	Process
NAND128W3A2BN	128M	SLC	10^5	90nm
HY27US08121A	512M	SLC	10^5	90nm
MT29F2G08AAD	2G	SLC	10^5	50nm
MT29F4G08AAC	4G	SLC	10^5	72nm
NAND08GW3B2C	8G	SLC	10^5	60nm
MT29F8G08MAAWC	8G	MLC	10^4	72nm
29F16G08CANC1	16G	SLC	10^5	50nm
MT29F32G08QAA	32G	MLC	10^4	50nm

Table 1: Devices tested

in this area includes a workshop report of our results [9] and an empirical characterization of flash memory carried out by Grupp et at. [12], analyzing performance of basic operations, power consumption, and reliability.

System-level studies have instead examined characteristics of entire flash-based storage systems, such as USB drives and SSDs. The most recent of these presents uFLIP [7], a benchmark for such storage systems, with measurements of a wide range of devices; this work quantifies the degraded performance observed for random writes in many such devices. Additional work in this area includes [14],[27], and [1]

Ben-Aroyo and Toledo [5] have presented detailed theoretical analyses of bounds on wear-leveling performance; however for realistic flash devices (i.e. with erase block size $>$ 1 page) their results show the existence of a bound but not its value.

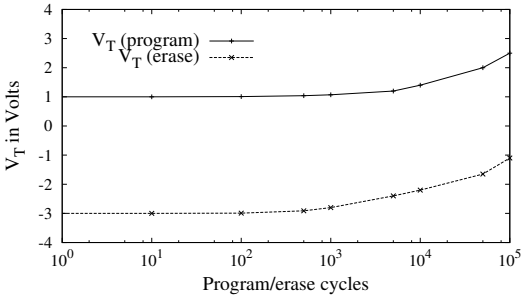


Figure 3: Typical V_T degradation with program/erase cycling for sub-90 nm flash cells. Data is abstracted from [24], [28], and [32].

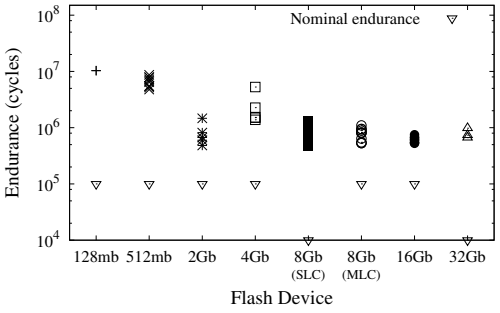


Figure 4: Write/Erase endurance by device. Each plotted point represents the measured lifetime of an individual block on a device. Nominal endurance is indicated by inverted triangles.

3 Experimental Results

3.1 Chip-level Endurance

Chip-level endurance was tested across a range of devices; more detailed results have been published in a previous workshop paper [9] and are summarized below.

Methodology: Flash chips were acquired both through distributors and by purchasing and disassembling mass-market devices. A programmable flash controller was constructed using software control of general-purpose I/O pins on a micro-controller to implement the flash interface protocol for 8-bit devices. Devices tested ranged from older 128Mbit (16MB) SLC devices to more recent 16Gbit and 32Gbit MLC chips; a complete list of devices tested may be seen in Table 1. Unless otherwise specified, all tests were performed at 25° C.

Endurance: Limited write endurance is a key characteristic of NAND flash—and all floating gate devices in general—which is not present in competing memory and storage technologies. As blocks are repeatedly erased and programmed the oxide layer isolating the gate degrades [23], changing the cell response to a fixed programming or erase step as shown in Figure 3. In practice this degradation is compensated for by adaptive programming and erase algorithms internal to the device, which use multiple program/read or erase/read steps to achieve the desired state. If a cell has degraded too much, however, the program or erase operation will terminate in an error; the external system must then consider the block *bad* and remove it from use.

Program/erase endurance was tested by repeatedly programming a single page with all zeroes (vs. the erased state of all 1 bits), and then erasing the containing block; this cycle was repeated until a program or erase operation terminated with an error status. Although nominal device endurance ranges from 10^4 to 10^5 program/erase cycles, in Figure 4 we see that the number of cycles until failure was higher in almost every case, often by nearly a factor of 100.

During endurance tests individual operation times were measured exclusive of data transfer, to reduce dependence on test setup; a representative trace is seen in Figure 5. The increased erase times and decreased program times appear to directly illustrate V_T degradation shown in Figure 3—as the cell ages it becomes easier to program and harder to erase, requiring fewer iterations of the internal write algorithm and more iterations for erase.

Additional Testing: Further investigation was performed to determine whether the surprisingly high en-

	Mean Endurance	Standard Deviation	Min. and max (vs. mean)
128mb	10.3 ($\times 10^6$)	0.003	+0.002 / -0.002
512mb	6.59	1.32	+2.09 / -1.82
2Gb	0.806	0.388	+0.660 / -0.324
4Gb	2.39	1.65	+2.89 / -1.02
8Gb SLC	0.827	0.248	+0.465 / -0.359
8Gb MLC*	0.783	0.198	+0.313 / -0.252
16Gb	0.614	0.078	+0.136 / -0.089
32Gb	0.793	0.164	+0.185 / -0.128

Table 2: Endurance in units of 10^6 write/erase cycles. The single outlier for 8 Gb MLC has been dropped from these statistics.

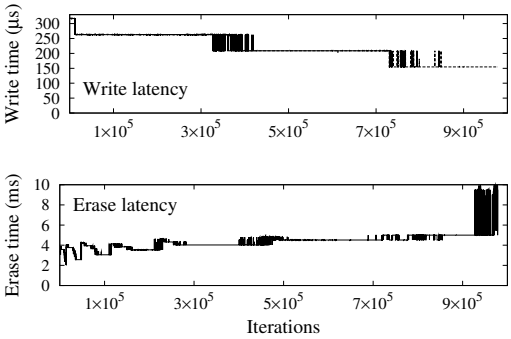


Figure 5: Wear-related changes in latency. Program and erase latency are plotted separately over the lifetime of the same block in the 8Gb MLC device. Quantization of latency is due to iterative internal algorithms.

durance of the devices tested is typical, or is instead due to anomalies in the testing process. In particular, we varied both program/erase behavior and environmental conditions to determine their effects. Due to the high variance of the measured endurance values, we have not collected enough data to draw strong inferences, and so report general trends instead of detailed results.

Usage patterns – The results reported above were measured by repeatedly programming the first page of a block with all zeroes (the programmed state for SLC flash) and then immediately erasing the entire block. Several devices were tested by writing to all pages in a block before erasing it; endurance appeared to decrease with this pattern, but by no more than a factor of two. Additional tests were performed with varying data patterns, but no difference in endurance was detected.

Environmental conditions – The processes resulting in flash failure are exacerbated by heat [32], although internal compensation is used to mitigate this effect [22]. The 16Gbit device was tested at 80° C, and no noticeable difference in endurance was seen.

Conclusions: The high endurance values measured were unexpected, and no doubt contribute to the measured performance of USB drives reported below, which

Device	Size	Chip Signature	USB ID
Generic	512Mbit	HY27US08121A	1976:6025
House	16Gbit	29F16G08CANC1	125F:0000
Memorex	4Gbit	MF12G2BABA	12F7:1A23

Table 3: Investigated devices

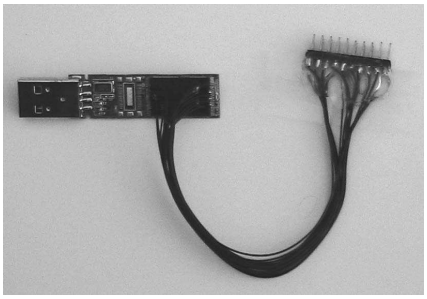


Figure 6: USB Flash drive modified for logic analyzer probing.

achieve high endurance using very inefficient wear-leveling algorithms. Additional experimentation is needed to determine whether these results hold across the most recent generation of devices, and whether flash algorithms may be tailored to produce access patterns which maximize endurance, rather than assuming it as a constant. Finally, the increased erase time and decreased programming time of aged cells bear implications for optimal flash device performance, as well as offering a predictive failure-detection mechanism.

3.2 FTL Investigation

Having examined performance of NAND flash itself, we next turn to systems comprising both flash and FTL. While work in the previous section covers a wide range of flash technologies, we concentrate here on relatively small mass-market USB drives due to the difficulties inherent in reverse-engineering and destructive testing of more sophisticated devices.

Methodology: we reverse-engineered FTL operation in three different USB drives, as listed in Table 3: *Generic*, an unbranded device based on the Hynix HY27US08121A 512Mbit chip, *House*, a MicroCenter branded 2GB device based on the Intel 29F16G08CANC1, and *Memorex*, a 512MB Memorex "Mini TravelDrive" based on an unidentified part.

In Figure 6 we see one of the devices with probe wires attached to the I/O bus on the flash chip itself. Reverse-engineering was performed by issuing specific logical operations from a Linux USB host (by issuing direct I/O reads or writes to the corresponding block device) and using an IO-3200 logic analyzer to capture resulting transactions over the flash device bus. From this captured

	Generic	House	Memorex
Structure	16 zones	4 zones	4 zones
Zone size	256 physical blocks	2048 physical blocks	1024 physical blocks
Free blocks list size	6 physical blocks per zone	30-40 physical blocks per zone	4 physical blocks per zone
Mapping scheme	Block-level	Block-level / Hybrid	Hybrid
Merge operations	Partial merge	Partial merge / Full merge	Full merge
Garbage collection frequency	At every data update	At every data update	Variable
Wear-leveling algorithm	Dynamic	Dynamic	Static

Table 4: Characteristics of reverse-engineered devices

data we were then able to decode the flash-level operations (read, write, erase, copy) and physical addresses corresponding to a particular logical read or write.

We characterize the flash devices based on the following parameters: *zone organization* (number of zones, zone size, number of free blocks), *mapping schemes*, *merge operations*, *garbage collection frequency*, and *wear-leveling algorithms*. Investigation of these specific attributes is motivated by their importance; they are fundamental in the design of any FTL [2, 3, 17, 19, 20, 21, 25], determining space requirements, i.e. the size of the mapping tables to keep in RAM (zone organization, mapping schemes), overhead/performance (merge operations, garbage collection frequency), device endurance (wear-leveling algorithms). The results are summarized in Table 4, and discussed in the next sections.

Zone organization: The flash devices are divided in zones, which represent contiguous regions of flash memory, with disjoint logical-to-physical mappings: a logical block pertaining to a zone can be mapped only in a physical block from the same zone. Since the zones function independently from each other, when one of the zones becomes unusable, other zones on the same device can still be accessed. We report actual values of zone sizes and free list sizes for the investigated devices in Table 4.

Mapping schemes: Block-mapped FTLs require smaller mapping tables to be stored in RAM, compared to page-mapped FTLs (Section 2.2). For this reason, the block-level mapping scheme is more practical and was identified in both Generic and multi-page updates of House flash drives. For single-page updates, House uses the simplified hybrid mapping scheme (which we will describe next), similar to Ban's NFTL [3]. The Memorex flash drive uses hybrid mapping: the data blocks are block-mapped and the log blocks are page-mapped.

Garbage collection: For the Generic drive, garbage collection is handled immediately after each write, eliminating the overhead of managing stale data. For House and Memorex, the hybrid mapping allows for several sequential updates to be placed in the same log block. Depending on specific writing patterns, garbage collection can have a variable frequency. The number of sequential updates that can be placed in a 64-page log block (before a new free log block is allocated to hold updated pages of the same logical block) ranges from 1 to 55 for Memorex and 1 to 63 for House.

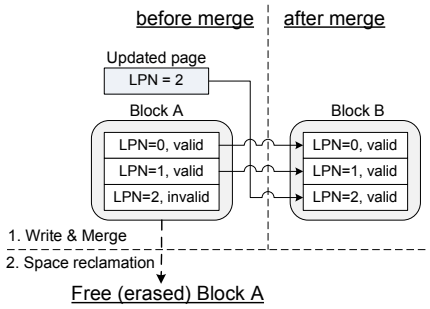


Figure 7: Generic device page update. Using block-level mapping and a partial merge operation during garbage collection. LPN = Logical Page Number. New data is merged with block A and an entire new block (B) is written.

We illustrate how garbage collection works after being triggered by a page update operation.

The Generic flash drive implements a simple page update mechanism (Figure 7). When a page is overwritten, a block is selected from the free block list, and the data to be written is merged with the original data block and written to this new block in a partial merge, resulting in the erasure of the original data block.

The House drive allows multiple updates to occur before garbage collection, using an approach illustrated in Figure 8. Flash is divided into two *planes*, even and odd (blocks B-even and B-odd in the figure); one log block can represent updates to a single block in the data area. When a single page is written, meta-data is written to the first page in the log block and the new data is written to the second page; a total of 63 pages may be written to the same block before the log must be merged. If a page is written to another block in the plane, however, the log must be merged immediately (via a full merge) and a new log started.

We observe that the House flash drive implements an optimized mechanism for multi-page updates, requiring 2 erasures rather than 4. This is done by eliminating the intermediary storage step in log blocks B-even and B-odd, and writing the updated pages directly to blocks C-

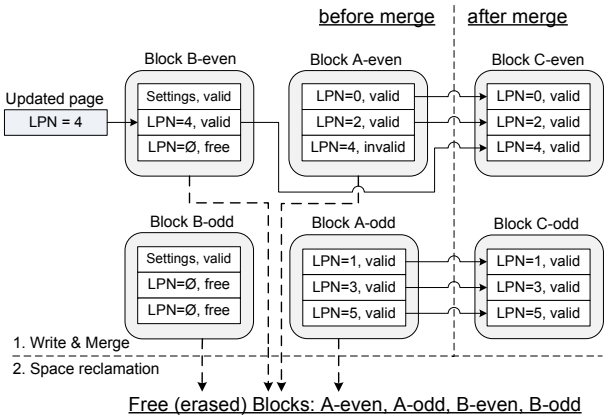


Figure 8: House device single-page update. Using hybrid mapping and a full merge operation during garbage collection. LPN = Logical Page Number. LPN 4 is written to block B, "shadowing" the old value in block A. On garbage collection, LPN 4 from block B is merged with LPNs 0 and 2 from block A and written to a new block.

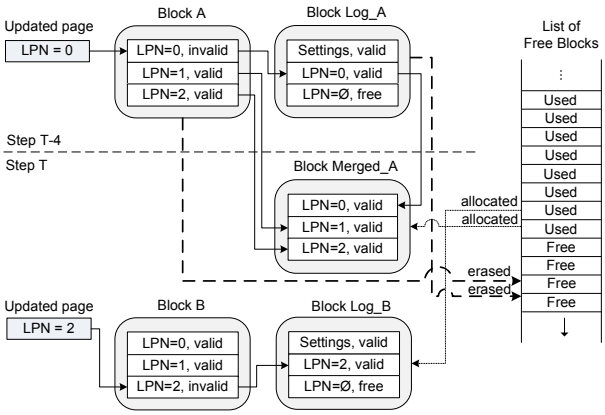


Figure 9: Memorex device page update. Using hybrid mapping and a full merge operation during garbage collection. LPN = Logical Page Number. LPN 2 is written to the log block of block B and the original LPN 2 marked invalid. If this requires a new log block, an old log block (Log_A) must be freed by doing a merge with its corresponding data block.

even and C-odd.

The Memorex flash drive employs a complex garbage collection mechanism, which is illustrated in Figure 9. When one or more pages are updated in a block (B), a merge is triggered if there is no active log block for block B or the active log block is full, with the following operations being performed:

- The new data pages together with some settings information are written in a free log block (Log_B).
- A *full merge* operation occurs, between two blocks (data block A and log block Log_A) that were accessed 4 steps back. The result is written in a free block (Merged_A). Note that the merge operation may be deferred until the log block is full.
- After merging, the two blocks (A and Log_A) are erased and added to the list of free blocks.

Wear-leveling aspects: From the reverse-engineered devices, static wear-leveling was detected only in the case of the Memorex flash drive, while both Generic and House devices use dynamic wear-leveling. As observed during the experiments, the Memorex flash drive is periodically (after every 138th garbage collection operation) moving data from one physical block containing rarely updated data, into a physical block from the list of free blocks. The block into which the static data has been moved is taken out of the free list and replaced by the rarely used block.

Conclusions: The three devices examined were found to have flash translation layers ranging from simple (Generic) to somewhat complex (Memorex). Our investigation provided detailed parameters of each FTL, including zone organization, free list size, mapping scheme, and static vs. dynamic wear-leveling methods. In combination with the chip-level endurance measurements presented above, we will demonstrate in Section 3.4 below the use of these parameters to predict overall device endurance.

3.3 Timing Analysis

Additional information on the internal operation of a flash drive may be obtained by timing analysis—measuring the latency of each of a series of requests and detecting patterns in the results. This is possible because of the disparity in flash operation times, typically 20μs, 200-300μs, and 2-4ms for read, write and erase respectively [9]. Selected patterns of writes can trigger differing sequences of flash operations, incurring different delays observable as changes in write latency. These changes offer clues which can help infer the following characteristics: (a) wear-leveling mechanism (static or dynamic) and parameters, (b) garbage collection mechanism, and (c) device end-of-life status.

Approach: Timing analysis uses sequences of writes to addresses $\{A_1, A_2, \ldots A_n\}$ which are repeated to provoke periodic behavior on the part of the device. The most straightforward sequence is to repeatedly write the same block; these writes completed in constant time for the Generic device, while results for the House device are seen in Figure 10. These results correspond to the FTL algorithms observed in Section 3.2 above; the Generic device performs the same block copy and erase for every write, while the House device is able to write to Block B (see Figure 8) 63 times before performing a merge operation and corresponding erase.

More complex flash translation layers require more

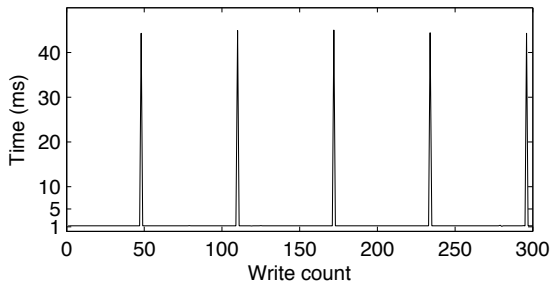


Figure 10: House device write timing. Write address is constant; peaks every 63 operations correspond to the merge operation (including erasure) described in Section 3.2.

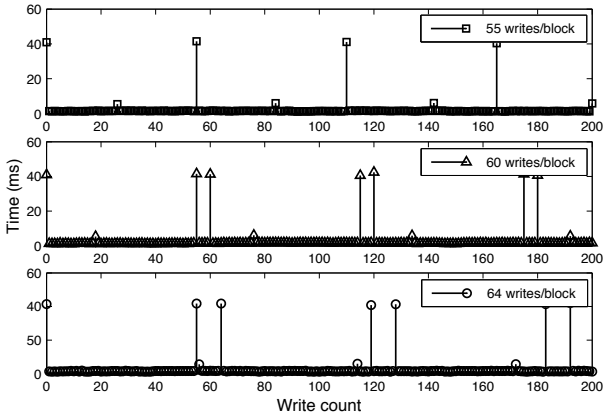


Figure 11: Memorex device garbage collection patterns. Access pattern used is $\{A_1 \times n, A_2 \times n, \ldots\}$ for $n = 55, 60, 64$ writes/block.

complex sequences to characterize them. The hybrid FTL used by the Memorex device maintains 4 log blocks, and thus pauses infrequently with a sequence rotating between 4 different blocks; however, it slows down for every write when the input stream rotates between addresses in 5 distinct blocks. In Figure 11 we see two patterns: a garbage collection after 55 writes to the same block, and then another after switching to a new block.

Organization: In theory it should be possible to determine the zones on a device, as well as the size of the free list in each zone, via timing analysis. Observing zones should be straightforward, although it has not yet been implemented; since each zone operates independently, a series of writes to addresses in two zones should behave like repeated writes to the same address. Determining the size of the free list, m, may be more difficult; variations in erase time between blocks may produce patterns which repeat with a period of m, but these variations may be too small for reliable measurement.

Wear-leveling mechanism: Static wear-leveling is indicated by combined occurrence of two types of peaks: smaller, periodic peaks of regular write/erase operations,

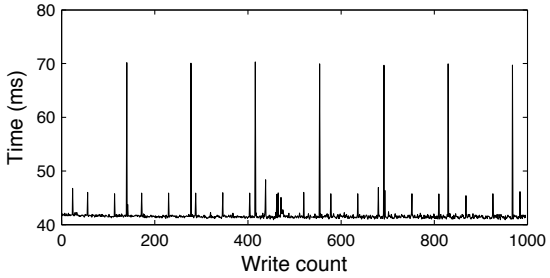


Figure 12: Memorex device static wear-leveling. Lower values represent normal writes and erasures, while peaks include time to swap a static block with one from the free list. Peaks have a regular frequency of one at every 138 write/erasure.

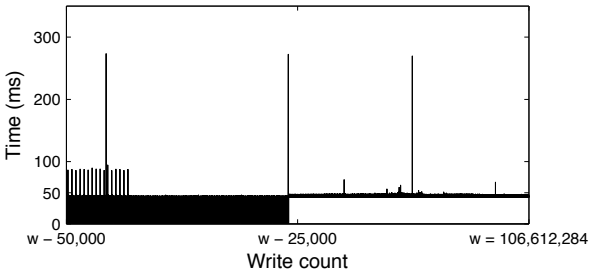


Figure 13: House device end-of-life signature. Latency of the final 5×10^4 writes before failure.

and higher, periodic, but less frequent peaks that suggest additional internal management operations. In particular, the high peaks are likely to represent moving static data into highly used physical blocks in order to uniformly distribute the wear. The correlation between the high peaks and static wear-leveling was confirmed via logic analyzer, as discussed in Section 3.2 and supported by extremely high values of measured device-level endurance, as reported in Section 3.3.

For the Memorex flash drive, Figure 12 shows latency for a series of sequential write operations in the case where garbage collection is triggered at every write. The majority of writes take approximately 45 ms, but high peaks of 70 ms also appear every 138th write/erase operation, indicating that other internal management operations are executed in addition to merging, data write and garbage collection. The occurrence of high peaks suggests that the device employs static wear-leveling by copying static data into frequently used physical blocks.

Additional tests were performed with a fourth device, *House-2*, branded the same as the House device but in fact a substantially newer design. Timing patterns for repeated access indicate the use of static wear-leveling, unlike the original House device. We observed peaks of 15 ms representing write operations with garbage collection, and higher regular peaks of 20 ms appearing at

Device	Parameters	Predicted endurance		Measured endurance
Generic	$m = 6, h = 10^7$	mh	6×10^7	$7.7 \times 10^7, 10.3 \times 10^7$
House	$m = 30, k = 64, h = 10^6$	between mh and mkh	between 3×10^7 and 1.9×10^9	10.6×10^7
Memorex	$z = 1024, k = 64, h = 10^6$(est.)	zkh	6×10^{10}	N/A

Table 5: Predicted and measured endurance limits.

approximately every 8,000 writes. The 5 ms time difference from common writes to the highest peaks is likely due to data copy operations implementing static wear-leveling.

End-of-life signature: Write latency was measured during endurance tests, and a distinctive signature was seen in the operations leading up to device failure. This may be seen in Figure 13, showing latency of the final 5×10^4 operations before failure of the House device. First the 80ms peaks stop, possibly indicating the end of some garbage collection operations due to a lack of free pages. At 25000 operations before the end, all operations slow to 40ms, possibly indicating an erasure for every write operation; finally the device fails and returns an error.

Conclusions: By analyzing write latency for varying patterns of operations we have been able to determine properties of the underlying flash translation algorithm, which have been verified by reverse engineering. Those properties include wear-leveling mechanism and frequency, as well as number and organization of log blocks. Additional details which should be possible to observe via this mechanism include zone boundaries and possibly free list size.

3.4 Device-level Endurance

By device-level endurance we denote the number of successful writes at logical level before a write failure occurs. Endurance was tested by repeated writes to a constant address (and to 5 constant addresses in the case of Memorex) until failure was observed. Testing was performed on Linux 2.6.x using direct (unbuffered) writes to the block devices.

Several failure behaviors were observed:

- **silent:** The write operation succeeds, but read verifies that data was not written.
- **unknown error:** On multiple occasions, the test application exited without any indication of error. In many casses, further writes were possible.
- **error:** An I/O error is returned by the OS. This was observed for the House flash drive; further write operations to any page in a zone that had been worn out failed, returning error.
- **blocking:** The write operation hangs indefinitely. This was encountered for both Generic and House flash drives, especially when testing was resumed after failure.

Endurance limits with dynamic wear-leveling: We measured an endurance of approximately 106×10^6 writes for House; in two different experiments, Generic sustained up to 103×10^6 writes and 77×10^6 writes, respectively. As discussed in Section 3.2, the House flash drive performs 4 block erasures for 1-page updates, while the Generic flash drive performs only one block erasure. However, the list of free blocks is about 5 times larger for House (see Table 3), which may explain the higher device-level endurance of the House flash drive.

Endurance limits with static wear-leveling: Wearing out a device that employs static wear-leveling (e.g. the Memorex and House-2 flash drives) takes considerably longer time than wearing out one that employs dynamic wear-leveling (e.g. the Generic and House flash drives). In the experiments conducted, the Memorex and House-2 flash drives had not worn out before the paper was submitted, reaching more than 37×10^6 writes and 26×10^8 writes, respectively.

Conclusions: The primary insight from these measurements is that wear-leveling techniques lead to a significant increase in the endurance of the whole device, compared to the endurance of the memory chip itself, with static wear-leveling providing much higher endurance than dynamic wear-leveling.

Table 5 presents a synthesis of predicted and measured endurace limits for the devices studied. We use the following notation:

N = total number of erase blocks,
k = total number of pages in the erase block,
h = maximum number of program/erase cycles of a block (i.e. the chip-level endurance),
z = number of erase blocks in a zone, and
m = number of free blocks in a zone.

Ideally, the device-level endurance is Nkh. In practice, based on the FTL implementation details presented in Section 3.2 we expect device-level endurance limits of mh for Generic, between mh and mkh for House, and zkh for Memorex. In the following computations, we use the program/erase endurance values, i.e. h, from Figure 4, and m and z values reported in Table 4. For Generic, $mh = 6 \times 10^7$, which approaches the actual measured values of 7.7×10^7 and 10.3×10^7. For House, $mh = 3 \times 10^7$ and $mkh = 30 \times 64 \times 10^6 = 1.9 \times 10^9$,

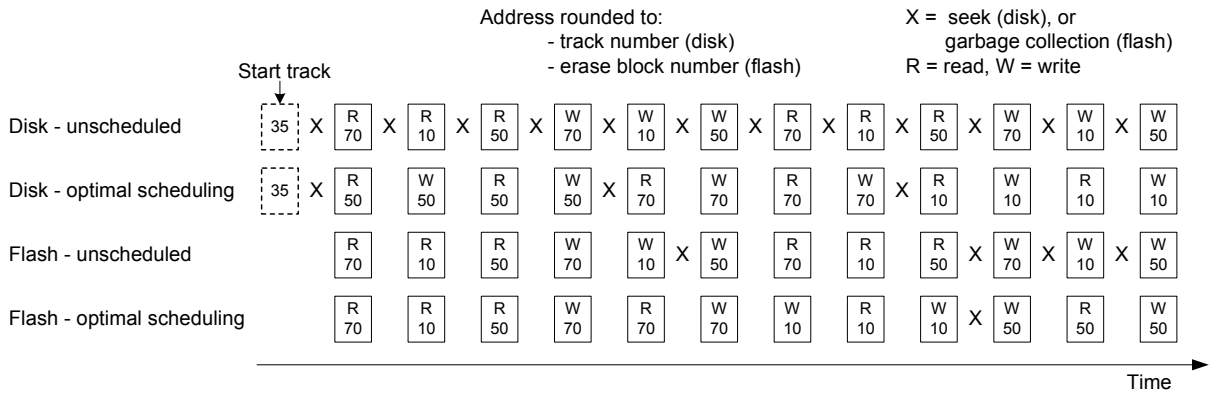


Figure 14: Unscheduled access vs. optimal scheduling for disk and flash. The requested access sequence contains both reads (R) and writes (W). Addresses are rounded to track numbers (disk), or erase block numbers (flash), and "X" denotes either a seek operation to change tracks (disk), or garbage collection to erase blocks (flash). We ignore the rotational delay of disks (caused by searching for a specific sector of a track), which may produce additional overhead. Initial head position (disk) = track 35.

with the measured device-level endurance of 10.6×10^7 falling between these two limits. For Memorex, we do not have chip-level endurance measurements, but we will use $h = 10^6$ in our computations, since it is the predominant value for the tested devices. We estimate the best-case limit of device-level endurance to be $zkh = 1024 \times 64 \times 10^6 \approx 6 \times 10^{10}$ for Memorex, which is about three orders of magnitude higher than for Generic and House devices, demonstrating the major impact of static wear-leveling.

3.5 Implications for Storage Systems

Space management: Space management policies for flash devices are substantially different from those used for disks, mainly due to the following reasons. Compared to electromechanical devices, solid-state electronic devices have no moving parts, and thus no mechanical delays. With no seek latency, they feature fast random access times and no read overhead. However, they exhibit asymmetric write vs. read performance. Write operations are much slower than reads, since flash memory blocks need to be erased before they can be rewritten. Write latency depends on the availability (or lack thereof) of free, programmable blocks. Garbage collection is carried out to reclaim previously written blocks which are no longer in use.

Disks address the seek overhead problem with scheduling algorithms. One well-known method is the elevator algorithm (also called SCAN), in which requests are sorted by track number and serviced only in the current direction of the disk arm. When the arm reaches the edge of the disk, its direction reverses and the remaining requests are serviced in the opposite order.

Since the latency of flash vs. disks has entirely different causes, flash devices require a different method than disks to address the latency problem. Request scheduling algorithms for flash have not yet been implemented in practice, leaving space for much improvement in this area. Scheduling algorithms for flash need to minimize garbage collection, and thus their design must be dependent upon FTL implementation. FTLs are built to take advantage of temporal locality; thus a significant performance increase can be obtained by reordering data streams to maximize this advantage. FTLs map successive updates to pages from the same data block together in the same log block. When writes to the same block are issued far apart from each other in time, however, new log blocks must be allocated. Therefore, most benefit is gained with a scheduling policy in which the same data blocks are accessed successively. In addition, unlike for disks, for flash devices there is no reason to reschedule reads.

To illustrate the importance of scheduling for performance as well as the conceptually different aspects of disk vs. flash scheduling, we look at the following simple example (Figure 14).

Disk scheduling. Let us assume that the following requests arrive: R 70, R 10, R 50, W 70, W 10, W 50, R 70, R 10, R 50, W 70, W 10, W 50, where R = read, W = write, and the numbers represent tracks. Initially, the head is positioned on track 35. We ignore the rotational delay of searching for a sector on a track. Without scheduling, the overhead (seek time) is 495. If the elevator algorithm is used, the requests are processed in the direction of the arm movement, which results in the following ordering: R 50, W 50, R 50, W 50, R 70, W 70, R 70, W 70, (arm movement changes direction), R 10, W 10, R 10, W 10. Also, the requests to the same track are grouped together, to minimize seek time; however, data integrity has to be preserved (reads/writes to the same disk track must be processed in the requested order, since

they might access the same address). This gives an overhead of 95, which is 5x smaller with scheduling vs. no scheduling.

Flash scheduling. Let us assume that the same sequence of requests arrives: R 70, R 10, R 50, W 70, W 10, W 50, R 70, R 10, R 50, W 70, W 10, W 50, where R = read, W = write, and the numbers represent erase blocks. Also assume that blocks are of size 3 pages, and there are 3 free blocks, with one block empty at all times. Without scheduling, 4 erasures are needed to accommodate the last 4 writes. An optimal scheduling gives the following ordering of the requests: R 70, R 10, R 50, W 70, R 70, W 70, W 10, R 10, W 10, W 50, R 50, W 50. We observe that there is no need to reschedule reads; however, data integrity has to be preserved (reads/writes to the same block must be processed in the requested order, since they might access the same address). After scheduling, the first two writes are mapped together to the same free block, next two are also mapped together, and so on. A single block erasure is necessary to free one block and accommodate the last two writes. The garbage collection overhead is 4x smaller with scheduling vs. no scheduling.

Applicability: Although we have explored only a few devices, some of the methods presented here (e.g. timing analysis) can be used to characterize other flash devices as well. FTLs range in complexity across devices; however, at low-end there are many similarities. Our results are likely to apply to a large class of devices that use flash translation layers, including most removable devices (SD, CompactFlash, etc.), and low-end SSDs. For high-end devices, such as enterprise (e.g. the Intel X25-E [16] or BiTMICRO Altima [6] series) or high-end consumer (e.g. Intel X25-M [15]), we may expect to find more complex algorithms operating with more free space and buffering.

As an example, JMicron's JMF602 flash controller [18] has been used for many low-end SSDs with 8-16 flash chips; it contains 16K of onboard RAM, and uses flash configurations with about 7% free space. Having little free space or RAM for mapping tables, its flash translation layer is expected to be similar in design and performance to the hybrid FTL that we investigated above.

At present, several flash devices including low-end SSDs have a built-in controller that performs wear-leveling and error correction. A disk file system in conjunction with a FTL that emulates a block device is preferred for compatibility, and also because current flash file systems still have implementation drawbacks (e.g. JFFS2 has large memory consumption and implements only write-through caching instead of write-back) [26].

Flash file systems could become more prevalent as the capacity of flash memories increases. Operating directly over raw flash chips, flash file systems present some advantages. They deal with long erase times in the background, while the device is idle, and use file pointers (which are remapped when updated data is allocated to a free block), thus eliminating the second level of indirection needed by FTLs to maintain the mappings. They also have to manage only one free space pool instead of two, as required by FTL with disk file systems. In addition, unlike conventional file systems, flash file systems do not need to handle seek latencies and file fragmentation; rather, a new and more suited scheduling algorithm as described before can be implemented to increase performance.

4 Analysis and Simulation

In the previous section we have examined the performance of several real wear leveling algorithms under close to worst-case conditions. To place these results in perspective, we wish to determine the maximum theoretical performance which any such on-line algorithm may achieve. Using terminology defined above, we assume a device (or zone within a device) consisting of N erase blocks, each block containing k separately writable pages, with a limit of h program/erase cycles for each erase block, and m free erase blocks. (i.e. the physical size of the device is N erase blocks, while the logical size is $N - m$ blocks.)

Previous work by Ben-Aroya and Toledo [5] has proved that in the typical case where $k > 1$, and with reasonable bounds on m, upper bounds exist on the performance of wear-leveling algorithms. Their results, however, offer little guidance for calculating these bounds. We approach the problem from the bottom up, using Monte Carlo simulation to examine achievable performance in the case of uniform random writes to physical pages. We choose a uniform distribution because it is both achievable (by means such as Ban's randomized wear leveling method [4]) and in the worst case unavoidable by any on-line algorithm, when faced with uniform random writes across the logical address space. We claim therefore that our numeric results represent a tight bound on the performance of any on-line wear-leveling algorithm in the face of arbitrary input.

We look for answers to the following questions:

- How efficiently can we perform *static wear leveling*? We examine the case where $k = 1$, thus ignoring erase block fragmentation, and ask whether there are on-line algorithms which achieve near-ideal endurance in the face of arbitrary input.
- How efficiently can we perform *garbage collection*? For typical values of k, what are the conditions needed for an on-line algorithm to achieve good performance

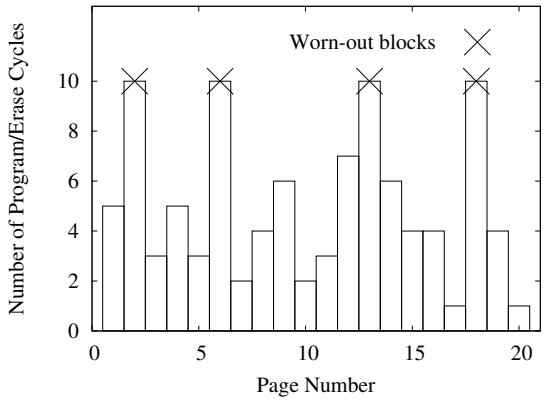


Figure 15: Trivial device failure ($N = 20$, $m = 4$, $h = 10$). Four blocks have reached their erase limit (10) after 100 total writes, half the theoretical maximum of Nh or 200.

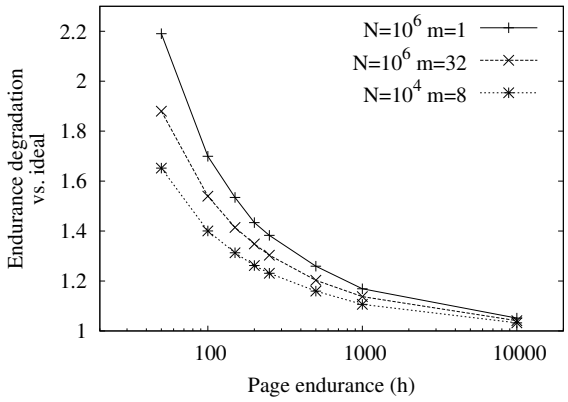


Figure 16: Wear-leveling performance. Endurance degradation (by simulation) for different numbers of erase blocks (N), block lifetime (h), and number of free blocks (m).

with arbitrary access patterns?

In doing this we use *endurance degradation* of an algorithm, or relative decrease in performance, as a figure of merit. We ignore our results on block-level lifetime, and consider a device failed once m blocks have been erased h times—at this point we assume the m blocks have failed, thus leaving no free blocks for further writes. In the perfect case, all blocks are erased the same number of times, and the drive endurance is $Nkh + mS$ (or approximately Nkh) page writes—i.e. the total amount of data written is approximately h times the size of the device. In the worst case we have seen in practice, m blocks are repeatedly used, with a block erase and reprogram for each page written; the endurance in this case is mh. The endurance degradation for an algorithm is the ratio of ideal endurance to achieved endurance, or $\frac{Nk}{m}$ for this simple algorithm.

4.1 Static Wear Leveling

As described in Section 2.2, *static wear leveling* refers to the movement of data in order to distribute wear evenly across the physical device, even in the face of highly non-uniform writes to the logical device. For ease of analysis we make two simplifications:

- Erase unit and program unit are of the same size, i.e. $k = 1$. We examine $k > 1$ below, when looking at garbage collection efficiency.
- Writes are uniformly distributed across physical pages, as described above.

Letting $X_1, X_2, \ldots X_N$ be the number of times that pages $1 \ldots N$ have been erased, we observe that at any point each X_i is a random variable with mean w/N , where w is the total number of writes so far. If the variance of each X_i is high and $m \ll N$, then it is likely that m of them will reach h well before $w = Nh$, where the expected value of each X_i reaches h. This may be seen in Figure 15, where in a trivial case ($N = 20$, $m = 4$, $h = 10$) the free list has been exhausted after a total of only $Nh/2$ writes.

In Figure 16 we see simulation results for a more realistic set of parameters. We note the following points:

- For $h < 100$ random variations are significant, giving an endurance degradation of as much as 2 depending on h and m.
- For $h > 1000$, uniform random distribution of writes results in near-ideal wear leveling.
- N causes a modest degradation in endurance, for reasonable values of N; larger values degrade endurance as they increase the odds that some m blocks will exceed the erase threshold.
- Larger values of m result in lower endurance degradation, as more blocks must fail to cause device failure.

For reasonable values of h, e.g. 10^4 or 10^5, these results indicate that randomized wear leveling is able to provide near-optimal performance with very high probability. However the implementation of randomization imposes its own overhead; in the worst case doubling the number of writes to perform a random swap in addition to every logical write. In practice a random block is typically selected every d writes and swapped for a block from the free list, reducing the overhead to $1/d$.

Although this reduces overhead, it also reduces the degree of randomization introduced. In the worst case—repeated writes to the same logical block—a page will remain on the free list until it has been erased d times before being swapped out. A page can thus only land in the free list h/d times before wearing out, giving performance equivalent to the case where the lifetime h' is h/d. As an example, consider the case where $d = 200$

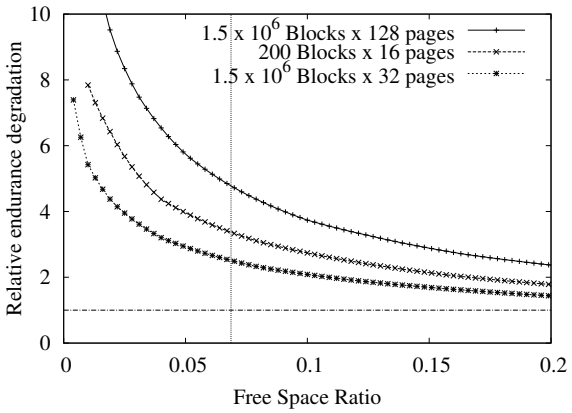


Figure 17: Degradation in performance due to wear-leveling for uniformly-distributed page writes. The vertical line marks a free percentage of 6.7%, corresponding to usage of 10^9 out of every 2^{30} bytes.

and $h = 10^4$; this will result in performance equivalent to $h = 50$ in our analysis, possibly reducing worst-case endurance by a factor of 2.

4.2 Garbage Collection

The results above assume an erase block size (k) of 1 page; in practice this value is substantially larger, in the devices tested above ranging from 32 to 128 pages. As a result, in the worst case m free pages may be scattered across as many erase blocks, and thus k pages must be erased (and $k - 1$ copied) in order to free a single page; however depending on the number of free blocks, the expected performance may be higher.

Again, we assume writes are uniformly and randomly distributed across Nk pages in a device. We assume that the erase block with the highest number of stale pages may be selected and reclaimed; thus in this case random variations will help garbage collection performance, by reducing the number of good pages in this block.

Garbage collection performance is strongly impacted by the utilization factor, or ratio of logical size to physical size. The more free blocks available, the higher the mean and maximum number of free pages per block and the higher the garbage collection efficiency. In Figure 17 we see the degradation in relative endurance for several different combinations of device size N (in erase blocks) and erase block size k, plotted against the fraction of free space in the device. We see that the worst-case impact of garbage collection on endurance is far higher than that of wear-leveling inefficiencies, with relative decreases in endurance ranging from 3 to 5 at a typical utilization (for low-end devices) of 93%.

Given non-uniform access patterns, such as typical file system access, it is possible that different wear-leveling strategies may result in better performance than the randomized strategy analyzed above. However, we claim that no on-line strategy can do better than randomized wear-leveling in the face of uniformly random access patterns, and that these results thus provide a bound on worst-case performance of any on-line strategy.

For an ideal on-line wear-leveling algorithm, performance is dominated by garbage collection, due to the additional writes and erases incurred by compacting partially-filled blocks in order to free up space for new writes. Garbage collection performance, in turn, is enhanced by additional free space and degraded by large erase block sizes. For example, with 20% free space and small erase blocks (32 pages) it is possible to achieve an endurance degradation of less than 1.5, while with 7% free space and 128-page blocks endurance may be degraded by a factor of 5.[1]

5 Conclusions

As NAND flash becomes widely used in storage systems, behavior of flash and flash-specific algorithms becomes ever more important to the storage community. Write endurance is one important aspect of this behavior, and one on which perhaps the least information is available. We have investigated write endurance on a small scale—on USB drives and on flash chips themselves—due to their accessibility; however the values we have measured and approaches we have developed are applicable across devices of all sizes.

Chip-level measurements of flash endurance presented in this work show endurance values far in excess of those quoted by manufacturers; if these are representative of most devices, the primary focus of flash-related algorithms may be able to change from wear leveling to performance optimization. We have shown how reverse-engineered details of flash translation algorithms from actual devices in combination with chip-level measurements may be used to predict device endurance, with close correspondence between those predictions and measured results. In addition, we have presented non-intrusive timing-based methods for determining many of these parameters. Finally, we have provided numeric bounds on achievable wear-leveling performance given typical device parameters.

Our results explain how simple devices such as flash drives are able to achieve high endurance, in some cases remaining functional after several months of continual testing. In addition, analytic and simulation results high-

[1] This is a strong argument for the new SATA *TRIM* operator [30], which allows the operating system to inform a storage device of free blocks; these blocks may then be considered free space by the flash translation layer, which would otherwise preserve their contents, never to be used.

light the importance of free space in flash performance, providing strong support for mechanisms like the TRIM command which allow free space sharing between file systems and flash translation layers. Future work in this area includes examination of higher-end devices, i.e. SSDs, as well as pursuing the implications for flash translation algorithms of our analytical and simulation results.

References

[1] AJWANI, D., MALINGER, I., MEYER, U., AND TOLEDO, S. Characterizing the performance of flash memory storage devices and its impact on algorithm design. In *Experimental Algorithms*. 2008, pp. 208–219.

[2] BAN, A. Flash file system. United States Patent 5,404,485, 1995.

[3] BAN, A. Flash file system optimized for page-mode flash technologies. United States Patent 5,937,425, 1999.

[4] BAN, A. Wear leveling of static areas in flash memory. United States Patent 6,732,221, 2004.

[5] BEN-AROYA, A., AND TOLEDO, S. *Competitive Analysis of Flash-Memory Algorithms*. 2006, pp. 100–111.

[6] BITMICRO NETWORKS. Datasheet: e-disk Altima E3F4FL Fibre Channel 3.5". available from `www.bitmicro.com`, Nov. 2009.

[7] BOUGANIM, L., JONSSON, B., AND BONNET, P. uFLIP: understanding flash IO patterns. In *Int'l Conf. on Innovative Data Systems Research (CIDR)* (Asilomar, California, 2009).

[8] CHUNG, T., PARK, D., PARK, S., LEE, D., LEE, S., AND SONG, H. System Software for Flash Memory: A Survey. In *Proceedings of the International Conference on Embedded and Ubiquitous Computing* (2006), pp. 394–404.

[9] DESNOYERS, P. Empirical evaluation of NAND flash memory performance. In *Workshop on Hot Topics in Storage and File Systems (HotStorage)* (Big Sky, Montana, October 2009).

[10] DRAMEXCHANGE. 2009 NAND flash demand bit growth forecast. `www.dramexchange.com`, 2009.

[11] GAL, E., AND TOLEDO, S. Algorithms and data structures for flash memories. *ACM Computing Surveys 37*, 2 (2005), 138–163.

[12] GRUPP, L., CAULFIELD, A., COBURN, J., SWANSON, S., YAAKOBI, E., SIEGEL, P., AND WOLF, J. Characterizing flash memory: Anomalies, observations, and applications. In *42nd International Symposium on Microarchitecture (MICRO)* (December 2009).

[13] GUPTA, A., KIM, Y., AND URGAONKAR, B. 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)* (Washington, DC, USA, 2009), ACM, pp. 229–240.

[14] HUANG, P., CHANG, Y., KUO, T., HSIEH, J., AND LIN, M. The Behavior Analysis of Flash-Memory Storage Systems. In *IEEE Symposium on Object Oriented Real-Time Distributed Computing* (2008), IEEE Computer Society, pp. 529–534.

[15] INTEL CORP. Datasheet: Intel X18-M/X25-M SATA Solid State Drive. available from `www.intel.com`, May 2009.

[16] INTEL CORP. Datasheet: Intel X25-E SATA Solid State Drive. available from `www.intel.com`, May 2009.

[17] INTEL CORPORATION. Understanding the flash translation layer FTL specification. Application Note AP-684, Dec. 1998.

[18] JMICRON TECHNOLOGY CORPORATION. JMF602 SATA II to Flash Controller. Available from `http://www.jmicron.com/Product_JMF602.htm`, 2008.

[19] KANG, J., JO, H., KIM, J., AND LEE., J. A Superblock-based Flash Translation Layer for NAND Flash Memory. In *Proceedings of the International Conference on Embedded Software (EMSOFT)* (2006), pp. 161–170.

[20] KIM, B., AND LEE, G. Method of driving remapping in flash memory and flash memory architecture suitable therefore. United States Patent 6,381,176, 2002.

[21] KIM, J., KIM, J. M., NOH, S., MIN, S. L., AND CHO, Y. A space-efficient flash translation layer for compactflash systems. *IEEE Transactions on Consumer Electronics 48*, 2 (2002), 366–375.

[22] KIMURA, K., AND KOBAYASHI, T. Trends in high-density flash memory technologies. In *IEEE Conference on Electron Devices and Solid-State Circuits* (2003), pp. 45–50.

[23] LEE, J., CHOI, J., PARK, D., AND KIM, K. Data retention characteristics of sub-100 nm NAND flash memory cells. *IEEE Electron Device Letters 24*, 12 (2003), 748–750.

[24] LEE, J., CHOI, J., PARK, D., AND KIM, K. Degradation of tunnel oxide by FN current stress and its effects on data retention characteristics of 90 nm NAND flash memory cells. In *IEEE Int'l Reliability Physics Symposium* (2003), pp. 497–501.

[25] LEE, S., SHIN, D., KIM, Y., AND KIM, J. 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 (SPEED)* (2008).

[26] MEMORY TECHNOLOGY DEVICES (MTD). Subsystem for Linux. JFFS2. Available from `http://www.linux-mtd.infradead.org/faq/jffs2.html`, January 2009.

[27] O'BRIEN, K., SALYERS, D. C., STRIEGEL, A. D., AND POELLABAUER, C. Power and performance characteristics of USB flash drives. In *World of Wireless, Mobile and Multimedia Networks (WoWMoM)* (2008), pp. 1–4.

[28] PARK, M., AHN, E., CHO, E., KIM, K., AND LEE, W. The effect of negative V_{TH} of NAND flash memory cells on data retention characteristics. *IEEE Electron Device Letters 30*, 2 (2009), 155–157.

[29] SANVIDO, M., CHU, F., KULKARNI, A., AND SELINGER, R. NAND flash memory and its role in storage architectures. *Proceedings of the IEEE 96*, 11 (2008), 1864–1874.

[30] SHU, F., AND OBR, N. Data Set Management Commands Proposal for ATA8-ACS2. ATA8-ACS2 proposal e07154r6, available from `www.t13.org`, 2007.

[31] SOOMAN, D. Hard disk shipments reach new record level. `www.techspot.com`, February 2006.

[32] YANG, H., KIM, H., PARK, S., KIM, J., ET AL. Reliability issues and models of sub-90nm NAND flash memory cells. In *Solid-State and Integrated Circuit Technology (ICSICT)* (2006), pp. 760–762.

Accelerating Parallel Analysis of Scientific Simulation Data via Zazen

Tiankai Tu,[1] Charles A. Rendleman,[1] Patrick J. Miller,[1] Federico Sacerdoti,[1]
Ron O. Dror,[1] and David. E. Shaw[1,2,3]

1. *D. E. Shaw Research, New York, NY 10036 USA*
2. *Center for Computational Biology and Bioinformatics, Columbia University, New York, NY 10032, USA*
3. *Corresponding author: David.Shaw@DEShawResearch.com*

Abstract

As a new generation of parallel supercomputers enables researchers to conduct scientific simulations of unprecedented scale and resolution, terabyte-scale simulation output has become increasingly commonplace. Analysis of such massive data sets is typically I/O-bound: many parallel analysis programs spend most of their execution time reading data from disk rather than performing useful computation. To overcome this I/O bottleneck, we have developed a new data access method. Our main idea is to cache a copy of simulation output files on the local disks of an analysis cluster's compute nodes, and to use a novel task-assignment protocol to co-locate data access with computation. We have implemented our methodology in a parallel disk cache system called *Zazen*. By avoiding the overhead associated with querying metadata servers and by reading data in parallel from local disks, Zazen is able to deliver a sustained read bandwidth of over 20 gigabytes per second on a commodity Linux cluster with 100 nodes, approaching the optimal aggregated I/O bandwidth attainable on these nodes. Compared with conventional NFS, PVFS2, and Hadoop/HDFS, respectively, Zazen is 75, 18, and 6 times faster for accessing large (1-GB) files, and 25, 13, and 85 times faster for accessing small (2-MB) files. We have deployed Zazen in conjunction with *Anton*—a special-purpose supercomputer that dramatically accelerates molecular dynamics (MD) simulations—and have been able to accelerate the parallel analysis of terabyte-scale MD trajectories by about an order of magnitude.

1 Introduction

Today, thousands of massively parallel computers are deployed around the world. The bountiful supply of computational power and the high-performance scientific simulations it has made possible, however, are not enough in themselves. To make scientific discoveries, the output from simulations must still be analyzed.

While simulation data are traditionally stored and accessed via parallel or network file systems, these systems have hardly kept up with the data deluge unleashed by faster supercomputers in the past decade [3, 28]. With terabyte-scale data quickly becoming the norm in many disciplines of computational science, I/O has become more critical a problem than ever.

A considerable amount of effort has gone into the design and implementation of special-purpose storage and middleware systems aimed at improving the I/O performance during a simulation [4, 5, 20, 22, 23, 25, 33]. By contrast, the I/O performance required in the course of analyzing the resulting data has received much less attention. From the viewpoint of overall time to solution, however, it is necessary to measure not only the time required to execute a simulation, but also the time required to analyze and interpret the output data. The I/O bottleneck *after* a simulation is thus as much an impediment to scientific discovery through advanced computing as the one that occurs *during* the simulation.

Our research aims to remove the analysis-time I/O impediment in a class of applications where the data output rate from a simulation is relatively low, yet the number of output files is relatively large. In particular, we focus on overcoming the data access bottleneck encountered by parallel analysis programs that execute on hundreds to thousands of processor cores and process millions to billions of simulation output files. Since the scale and complexity of this class of data-intensive analysis applications preclude the use of conventional storage systems, which have already struggled to handle less demanding I/O workloads, we introduce a new data access method designed to achieve a much higher level of performance.

Our solution works as follows. During a simulation, results are saved incrementally in a series of files. We instruct the I/O node of a parallel supercomputer not only to write each output file to a parallel/network file server, but also to send the content of the file to some node of a separate cluster that is dedicated to post-simulation data analysis. We refer to such a cluster as an *analysis cluster* and its nodes as *analysis nodes*. Our goal is to distribute the output files evenly among the analysis nodes. Upon receiving the data from the I/O

node, an analysis node caches (i.e., stores) the content as a local copy of the file. Each analysis node manages only the files it has cached locally. No metadata, either centralized or distributed, are maintained to keep track of which node has cached which files. When a simulation is completed, its (many) output files are stored on the file server as well as distributed (more or less) evenly among all analysis nodes.

At analysis time, each process of a parallel analysis program (assuming one process per analysis node) determines which files have been cached locally, and uses this knowledge to participate in the execution of a distributed task-assignment protocol (in collaboration with processes of the analysis program running on other analysis nodes). The outcome of the protocol is an assignment (i.e., a partitioning) of the file I/O tasks, in such a way that each file of a simulation dataset will be read by one and only one process (for correctness), and that each process will be mostly responsible for reading the files that have been cached locally (for efficiency). After completing the protocol execution, all processes proceed in parallel without further communication to coordinate I/O. (They may still communicate with one another for other purposes.) To retrieve each assigned file, a process first attempts to read it from the local disks, and then in case of a local cache miss, fetches the file from the parallel/network file system on which the entire simulation output dataset is persistently stored.

We have implemented our methodology in a parallel disk cache system called *Zazen* that has three components: (1) a disk cache server that runs on every compute node of an analysis cluster and manages locally cached data, (2) a client library that provides API functions for operating the cache, and (3) a communication library that queries the cache and executes the task-assignment protocol, referred to as *the Zazen protocol*.

Experiments show that Zazen is scalable, efficient, and robust. On a Linux cluster with 100 nodes, executing the Zazen protocol to assign I/O tasks for one billion files takes less than 15 seconds. By avoiding the overhead associated with querying metadata servers and by reading data in parallel from local disks, Zazen delivers a sustained read bandwidth of more than 20 gigabytes per second on 100 nodes when reading large (1-GB) files. It is 75 times faster than NFS running on a high-end enterprise storage server, and 18 and 6 times faster, respectively, than PVFS2 [8, 31] and Hadoop/HDFS [15] running on the same 100 nodes. When reading small (2-MB) files, Zazen achieves a sustained read performance of about 8 gigabytes per second on 100 nodes, outperforming NFS, PVFS2, and Hadoop/HDFS by a factor of 25, 13, and 85, respectively. We emphasize that despite its large performance advantage over network/parallel file systems, Zazen serves only as a cache system to improve parallel file read speed. Without a slower but more reliable file system as backup, Zazen would not be able to handle cache misses. Finally, our experiments demonstrate that Zazen works even when up to 50% of the nodes have gone offline. The only noticeable effect is a slowdown in execution time, which degrades gracefully, as predicted by our failure model.

We have deployed Zazen in conjunction with Anton [38]—a special-purpose supercomputer developed at D. E. Shaw Research for molecular dynamics (MD) simulations—to support the parallel analysis of terabyte-scale MD trajectories. Compared with the performance of implementations that access data from a high-end NFS server, the end-to-end execution time of a large number of parallel trajectory analysis programs that access data via Zazen has improved by about an order of magnitude.

2 Background

Scientific simulations seek numerical approximations of solutions to the partial differential, ordinary differential, algebraic, integral, or particle equations that govern the physical systems of interest. The solutions, typically computed as displacements, pressures, temperatures, or other physical quantities associated with grid points, mesh nodes, or particles, represent the states of the system being simulated and are stored to disk.

Time-dependent simulations such as mantle convection, supernova explosion, seismic wave propagation, and bio-molecular motions output a series of solutions, each representing the state of the system at a particular simulated time. We refer to these solutions as *output frames* or simply *frames*. While the organization of frames on disk is application-dependent, we assume in this paper that all frames are of the same size and each is stored in a separate file.

An important class of time-dependent simulations has the following characteristics. First, they output *a large number of small frames*. A millisecond-scale MD simulation, for example, may generate millions to billions of frames, each having a size less than a few megabytes. Second, the frames are *write once read many*. Once a frame is generated and stored to disk, it is usually read multiple times by data analysis programs. A frame, for all practical purposes, is never modified unless deleted. Third, *unique integer sequence numbers* can be used to distinguish the frames, which are generated in a temporal order as a simulation marches forward in time. Fourth, frames are *amenable to parallel processing* at analysis time. For example, our recent work [46] has demonstrated how to use the MapReduce programming model to access frames in an arbitrary order in the map phase and restore their temporal order in the reduce phase.

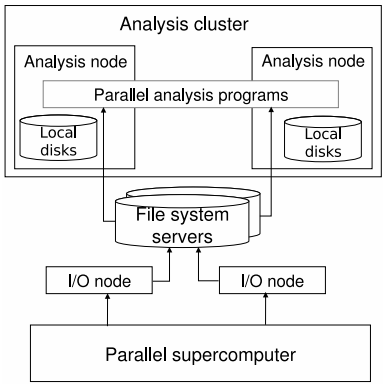


Figure 1: Simulation I/O infrastructure. Parallel analysis programs traditionally read simulation output from a parallel or network file system.

Traditionally, frames are stored and accessed via a parallel or network file system, as shown in Figure 1. At the bottom of the figure lies a parallel supercomputer that executes scientific simulations and outputs data through I/O nodes, which are specialized service nodes for tightly coupled parallel machines such as IBM's BlueGene, Cray's XT series, or Anton. These nodes aggregate the data generated by the compute nodes within a supercomputer and store the results to the file system servers. Two I/O nodes are shown in Figure 1 for illustration purposes; the actual number of I/O nodes varies by system. The top of Figure 1 shows an analysis cluster may or may not be co-located with a parallel supercomputer. In the latter case, simulation data can be stored to file servers close to the analysis cluster—either online, using techniques such as ADIO [12, 43] and PDIO [30, 40] or offline, using high-performance data transfer tools such as GridFTP [14]. An analysis cluster is typically much smaller in scale than a parallel supercomputer and has on the order of tens to hundreds of analysis compute nodes. While an analysis cluster provides tremendous computational and memory resources to parallel analysis programs, it also imposes intensive I/O workload to the underlying file servers, which, in most cases, cannot keep up.

3 Solution Overview

The local disks on the analysis nodes, shown in Figure 1, are typically unused except for storing operating systems files and temporary user data. While an individual analysis node may have much smaller disk space than file servers, the aggregated capacity of all local disks in an analysis cluster may be on par with or even exceed that of the file servers. With such abundant and potentially useful storage resources at our disposal, it is natural to ask how we can exploit these resources to solve the problem of reading a large number of frames in parallel.

3.1 The Main Idea

Our main idea is to cache a copy of each output frame in the local disks of arbitrary analysis nodes, and use a data location–aware task-assignment protocol to coordinate the parallel read of the cached data at analysis time.

Because simulation frames are *write once read many*, cache consistency is guaranteed. Thus, at simulation time, we arrange for the I/O nodes of a parallel supercomputer to push a copy of output frames to the local disks of the analysis nodes as the frames are generated and stored to a file server. We cache each frame on one and only one node and place consecutive frames on different nodes for load balancing. The assignment of frames to nodes can be arbitrary as long as the frames are spread across the analysis nodes more or less evenly. We choose a first machine randomly from a list of known analysis nodes and push frames to that machine and then its peers in a round-robin order. When caching frames from a long-running simulation that lasts for days or weeks, some of the analysis nodes will inevitably crash and become unavailable. We detect and skip the crashed nodes and place the output frames on the surviving nodes. Note that we do not use a metadata server to keep track of where frames are cached.

When executing a parallel analysis program, we use a cluster resource manager such as SLURM [39, 49] to obtain as many analysis nodes as available. We instruct each process to read frames directly from its local disk cache. To coordinate the parallel read of the cached frames and to ensure that each frame is read by one and only one node, we execute a data location–aware task-assignment protocol before performing any I/O. The purpose of this protocol is to co-locate data access with computation. Upon completion of the protocol execution, each process receives a list of integer sequence numbers that correspond to the frames it is responsible for reading. Most, if not all, of the assigned frames are those that have been cached locally. Those that are missing from the cache—for example, those that are cached on a crashed node or those that have been evicted—are fetched from the file servers and then cached in local disks.

3.2 Applicability

The proposed solution works only if the aggregated disk space of the dedicated analysis cluster is large enough to accommodate tens to hundreds of terabyte-scale simulation output datasets, so that recently cached datasets are not evicted too quickly. Considering the density and the price of today's hard drives, we expect that it is both technologically and economically feasible to provision a medium-size cluster with hundreds of terabytes to a few petabytes of disk storage. As an example, the cluster at Intel Research Pittsburgh, which is part of the

OpenCirrus consortium, is reported to have 150 nodes with over 400 TB of disk storage [18].

Another prerequisite of our solution is that the data output rate from a simulation is relatively low. In practice, this means that the data output rate must be lower than both the network bandwidth to and the disk bandwidth on any analysis node. If this is true, we can use multithreading techniques to overlap data caching with computation and avoid slowing down the execution of a simulation.

Certain classes of simulations cannot take advantage of the proposed caching mechanism because of the restrictions imposed by these two prerequisites. Nevertheless, many time-dependent simulations do satisfy both prerequisites and are amenable to simulation-time data caching.

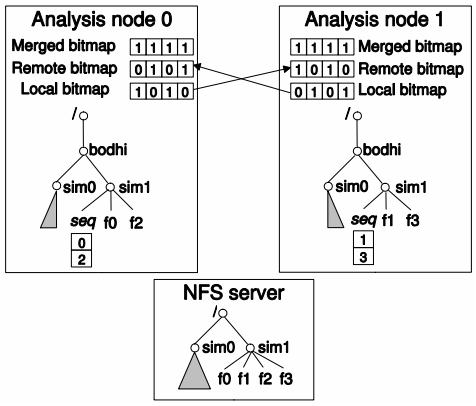


Figure 2: Simulation data organization. Frames are stored to file servers as well as the analysis nodes.

3.3 An Example

We assume that an analysis cluster has only two nodes as shown in Figure 2. We use the local disk partition mounted at `/bodhi` as the cache space. We also assume that an MD simulation generates four frames named `f0`, `f1`, `f2`, and `f3` in a directory `/sim1/`. As the frames are generated by the simulation at certain intervals and pushed to an NFS server, they are also stored to nodes 1 and 2 in an alternating fashion, with `f0` and `f2` going to node 1, and `f1` and `f3` to node 2. When a node receives an output file, it prepends the local disk cache root, that is, `/bodhi`, to the full path name of the file, creates a cache file locally using the derived file name (e.g., `/bodhi/sim1/f0`), and writes the contents. After the data is cached locally, a node records the sequence number of the frame—which is sent by an I/O node—in a *sequence log file* that is stored in the local directory along with the frames.

Figure 2 shows the data organization on the NFS server and on the two analysis nodes. The isosceles triangles represent datasets that have already been stored on the NFS server at directory `/sim0/`; the right triangles represent the portions of files that have been cached on nodes 0 and 1, respectively. The `seq` file represents the sequence log file that is created and updated independently on each node.

When analyzing the dataset stored at `/sim1`, we open its associated sequence log file (i.e., `/bodhi/sim1/seq`) on each node in parallel, and retrieve the sequence numbers of the frames that have been cached locally. We then construct a bitmap with four entries (equal to the number of frames to be analyzed) and set the bits for those that it has cached locally. On node 0, the first and third bits are set; on node 1, the second and fourth bits.

We then exchange the bitmaps between the nodes. By examining the combined results, both nodes realize that that all requested frames have been cached somewhere in the analysis cluster. Since node 0 has local access to `f0` and `f2`, it signs up for reading these two frames—with the knowledge that the other node must have local access to the remaining two files. Node 1 makes a similar decision and signs up for `f1` and `f3`. Both nodes then proceed in parallel and read the cached frames without further communication. Because all requested frames have been cached on either node 0 or node 1, no read requests are sent to the NFS server.

With only two nodes in this example, converting local disks to a distributed cache might not appear to be worthwhile. Nevertheless, when hundreds or more nodes are present, the effort pays off as it allows us to harness the vast storage capacities and I/O bandwidths distributed across many nodes.

3.4 Implementation

We have implemented our methodology in a parallel disk cache system called *Zazen*. The literal meaning of Zazen is "enlightenment through seated meditation." By a stretch of imagination, we use the term to describe the behavior of the analysis nodes in an anthropomorphic way: Instead of consulting a master node for advice on what data to read, every node seeks its inner knowledge of what has been cached locally to help decide its own action, thereby becoming "enlightened."

As shown in Figure 3, the Zazen system consists of three components:

- The *Bodhi library*: a client library that provides API functions (open, write, read, query, and close) for I/O nodes of parallel supercomputers to push output frames to analysis nodes, and for parallel analysis programs to query and read data from *local* disks.
- The *Bodhi server*: a disk cache server that manages the frames that have been cached on local disks and provides read service to local clients and write

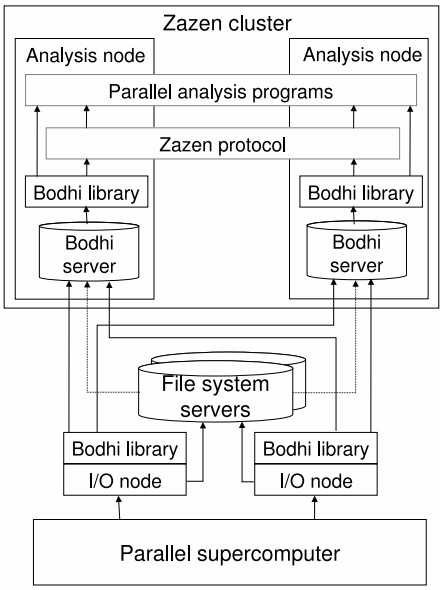


Figure 3: Overview of the Zazen system. The Bodhi library provides API functions for operating the local disk caches. The Bodhi server manages the frames cached locally and services client requests. The Zazen protocol coordinates parallel read of the cached data.

service to remote clients.

- The *Zazen protocol*: a data location–aware task-assignment protocol for assigning frame read tasks to analysis nodes.

We refer to the distributed local disks collectively as the *Zazen cache* and the hosting analysis cluster as the *Zazen cluster*. The Zazen cluster supports two types of applications: *writers* and *readers*. Writers are I/O processes running on the I/O nodes of a supercomputer. They only write output frames to the Zazen cache and never read them back. Readers are parallel processes of an analysis program. They run on the analysis nodes, execute the Zazen protocol, read data from local disk caches, and, in case of cache misses, have data fetched (by Bodhi servers) into the Zazen cache. As shown in Figure 3, inter-processor communication takes place only at the application level and the Zazen protocol level. The Bodhi library and server on different nodes do not communicate with one another directly as they do not share information with respect to which frames have been cached locally.

When frames are stored in the Zazen cache, they are treated as either *natives* or *aliens*. A native frame is one that is written to the Zazen cache by an I/O node that calls the Bodhi library write function. An alien frame is one that is brought into the Zazen cache by a Bodhi server because of a local cache read miss; it is the by-product of a call to the Bodhi library read function. Note that a frame can be a native on at most one node, but can be an alien on multiple nodes. To distinguish the two types of cached frames, we maintain two sequence log files for each simulation dataset to keep track of the integer sequence numbers of the native and alien frames, respectively. (The example of Section 3.2 showed only the native sequence log files.)

While the Bodhi library and server provide the necessary machinery for operating the Zazen cache, the intelligence of coordinating the parallel read of the cached data—the core of our innovation—lies in the Zazen protocol.

4 The Zazen Protocol

At first glance, it might appear that the coordination of the parallel read from the Zazen cache is unnecessary. Indeed, if no node would ever fail and cached data were never evicted, every node could simply consult its native sequence log file (associated with a particular dataset) and read the frames it has cached locally. Because an I/O node stores each output frame to one and only one node, neither duplicate reads nor cache read misses would occur.

Unfortunately, the premise of this argument is rarely true in practice. Analysis nodes do fail in various unpredictable ways due to hardware, software, and human errors. If a node crashes for some reason other than disk failures, the frames cached on that node become temporarily unavailable. Assume that during the node's down time, a parallel analysis code requests access to a dataset that has been partially cached on the failed node. Furthermore, assume that under the auspices of some oracle, the surviving analysis nodes are able to decide who should read which missing frames. Then the missing frames are fetched from the file servers and—as an intended side effect—cached locally on the surviving nodes as aliens. Assume that after the execution of the analysis, the failed node recovers and is back online. All of its locally cached frames once again become available. If the previously accessed dataset is processed again, some of its frames are now cached twice: once on the recovered node (as natives) and once on some other nodes (as aliens). More complex failure and recovery sequences may take place, which can lead to a single frame being cached multiple times or not cached at all.

We devised the Zazen protocol to guarantee that regardless how many (i.e., zero or more) copies of a frame have been cached, it is read by one and only one node. To achieve this goal, we enforce the following rules in order:

- Rule (1): If a frame is cached as a native on some node, we use that node to read the frame.
- Rule (2): If a frame is not cached as a native on any node and is cached as an alien once on some node,

we use that node to read the frame.

- Rule (3): If a frame is missing from the cache, we choose an arbitrary node to read the frame and cache the file.

We define a frame as *missing* if either the frame is not cached at all on any node or the frame is not cached as a native but is cached as an alien multiple times on different nodes.

The rationale behind the rules is as follows. Each frame is cached as a native *once and only once* on one of the analysis nodes when the frame file is pushed into the Zazen cache by an I/O node. If a native copy exists, it becomes an undisputed sole winner and knocks off other competitors who offer to provide an alien copy. Otherwise, a winner emerges only if it is the sole holder of an alien copy. If multiple alien copies exist, all contenders back off to avoid expensive distributed arbitration. An arbitrary node is then chosen to service the frame.

To coordinate the parallel read of cached data, all processes of a parallel analysis program must execute the Zazen protocol by calling an API function named `zazen`. The input to the `zazen` function includes `bodhi` (a handle to the local cache), `simdir` (the base directory of a simulation dataset), `begin` (the sequence number of the first frame to be accessed), `end` (the sequence number of the last frame to be accessed), and `stride` (the stride between the frames to be accessed). The output of the `zazen` function is an abstract data type `zazen_bitmap` that contains the necessary information for each process to find out which frames of the dataset it should read. Because the order of parallel accessing of frames is irrelevant, as explained in Section 2, each process consults the `zazen_bitmap` and calls the Bodhi library read function to read the frames it is responsible for processing, in parallel with other processes.

The main techniques we used to implement the Zazen protocol are *bitmaps* and *all-to-all reduction algorithms* [6, 11, 44]. The former provides a compact data structure for recording the presence or non-presence of frames, which may number in the billions. The latter furnishes an efficient mechanism for performing interprocessor collective communications. While we could have implemented all-to-all reduction algorithms from scratch (with a fair amount of effort), we chose instead to use an MPI library [26] as it already provides an optimized implementation that scales on to tens of thousands of nodes. In what follows, we simplify the description of the Zazen protocol algorithm by assuming that only one process (of a parallel analysis program) runs on each node.

1. *Creation of local native bitmaps.* Each process calls the Bodhi library query function to obtain the sequence numbers of the frames that have been cached as native on the local node. It creates an empty bitmap, whose number of bits is equal to the total number of frames to be accessed. Next, it sets the bits corresponding to the sequence numbers of the locally cached natives and produces a partially filled bitmap called a *local native bitmap*.
2. *Generating of global native bitmaps.* All the processes perform an all-to-all reduction that applies a bitwise-or operation on the local native bitmaps. On return, each node obtains an identical new bitmap called a *global native bitmap* that represents *all* the frames that have been cached as natives somewhere.
3. *Identification of local native reads.* Each process checks if the global native bitmap is fully set. If so, we have a perfect native cache hit ratio of 100%. The Zazen protocol is completed and every process proceeds to read the frames specified in its local native bitmap, knowing that the remaining frames are being read by other processes. Otherwise, some frames are not cached as natives, though they may well exist on some nodes as aliens.
4. *Creation of local alien bitmaps.* Each process queries its local Bodhi server for a second time to find the sequence numbers of the frames that are cached as aliens. It creates a new empty bitmap that uses *two bits*—instead of just one bit for the case of local native bitmaps—for each frame. The low-order (rightmost) bit is used in this step and the high-order (leftmost) bit will be used in the next step. Initially, both bits are set to 0. A process checks whether the sequence number of each of its locally cached aliens is already set in the global native bitmap. If so, the process ignores the local alien copy to enforce Rule (1). Otherwise, the process uses the alien copy's sequence number as an index to locate the corresponding frame entry in the new bitmap and sets the low-order bit to one.
5. *Generation of global alien bitmaps.* All the processes perform a second round of all-to-all reduction to combine the contributions from local alien bitmaps. Given a pair of input two-bit entries, we generate an output two-bit entry by applying a commutative operator denoted as "$\circ$" that works as follows:

   $$00 \circ xx \rightarrow xx,\ 10 \circ \mathrm{xx} \rightarrow 10, \text{ and } 01 \circ 01 \rightarrow 10\ ,$$

   where x stands for either 0 or 1. Note that an input two-bit entry can never be 11 and the high-order bit of the output is set to one only if both input bitmaps have their lower-order bits set (i.e., claiming to have cached the frame as an alien). On return, each process receives an identical new bitmap called a

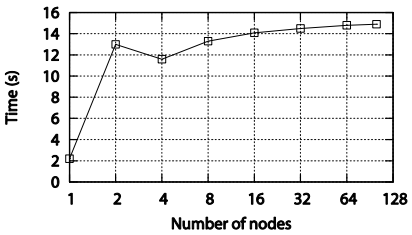


Figure 4: Fixed-problem-size scalability. The execution time of the Zazen protocol for processing one billion frames grows only marginally as the number of analysis nodes increases from 1 to 100.

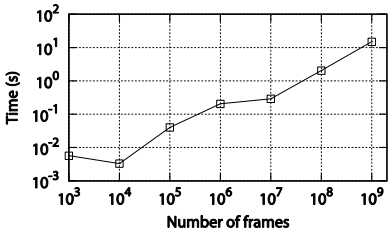


Figure 5: Fixed-cluster-size scalability. The execution time of the Zazen protocol on 100 nodes grows sub-linearly with the number of frames.

global alien bitmap that records the frames that have been cached as aliens.

6. *Identification of local alien reads.* Each process performs a bitwise-and operation on its local alien bitmap and the global alien bitmap. It identifies the offsets of the non-zero entries (which must be 01) of the result to enforce Rule (2). Those entries represent the frames for which the process is the sole alien-copy holder. Together, the identified local native and alien reads represent the frames a process voluntarily signs up to read.
7. *Adoption of residue frames.* Each process conducts a bitwise-or operation on the global native bitmap and the low-order bits of the global alien bitmap. The unset bits in the output bitmap are *residue* frames for which no process has signed up. A frame may be a residue for one of the following reasons: (1) it has been cached on a crashed node, (2) it has been cached multiple times as an alien but not once as a native, or (3) it has been evicted from the cache. Regardless of the cause, the residues are treated by all processes as the elements of a single array. Each process then executes a partitioning algorithm, in parallel without communication, to divide the array into contiguous blocks and adopt the block that corresponds to its rank among all the processes.

The Zazen protocol has two distinctive features. First, the data location information is obtained directly on each node—an embarrassingly parallel and scalable operation—rather than returned by a metadata server or servers. Second, if a node crashes, the protocol still works. The frames cached on the failed node are simply treated as cache misses.

5 Performance Evaluation

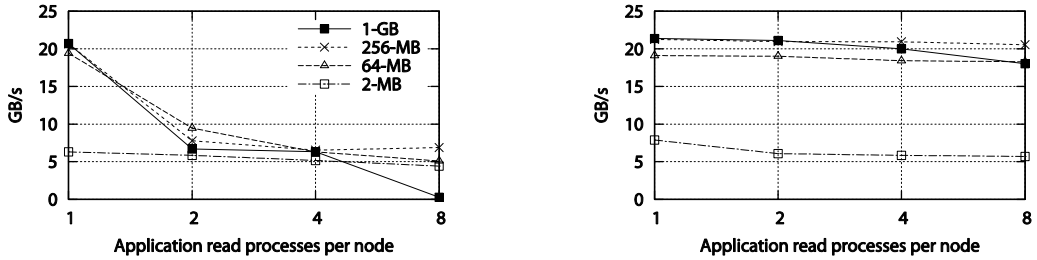
We have evaluated the scalability, efficiency, and robustness of Zazen on a commodity Linux cluster with 100 nodes that are hosted in three racks. The nodes are interconnected via a 1-gigabit Ethernet with full bisectional bandwidth. Each node runs CentOS 4.6 with a kernel version of 2.6.26 and has two Intel Xeon 2.33-GHz quad-core processors, 16 GB physical memory, and four 500-GB 7200-RPM SATA disks. We organized the local disks as a software RAID 0 (striped) partition and managed the RAID volume with an ext3 file system. The usable local disk cache space on each node is about 1.8 TB; so the total capacity of the Zazen cache is 180 TB. All nodes have access to common NFS directories exported by a number of enterprise storage servers. Evaluation programs were written in C unless otherwise specified.

5.1 Scalability

Because the Bodhi client and server are standalone components that can be deployed on as many nodes as available, they are inherently scalable. Hence, the scalability of the Zazen system, as a whole, is essentially determined by that of the Zazen protocol.

In the following experiments, we measured how the execution time of the Zazen protocol scales as we increased the cluster size and the problem size, respectively. No files were physically generated, stored to, or accessed from the Zazen cache. To create local bitmaps without querying local Bodhi servers (since no files actually existed in this particular test) and to force the execution of the optional second round of all-to-all reduction (for generating global alien bitmaps), we modified the procedure outlined in Section 4 so that each process set a non-overlapping, contiguous sequence of n/p frames as natives, where n is the total number of frames and p is the number of analysis nodes. The rest of the frames were treated as aliens. The MPI library used in these experiments was Open MPI 1.3.2 [26].

Figure 4 shows the execution time of the Zazen protocol for assigning one billion frames as the number of analysis nodes increases from 1 to 100. Each data point presents the average of three runs whose coefficient of variation (standard deviation over mean) is negligible. The execution time on one node is the time for manipulating the bitmaps locally and does not include any communication overhead. The dip of the curve in the four-node case may have been caused by the MPI runtime choosing a different optimized `MPI_Allreduce`



(a) One Bodhi read daemon per application read process

(b) One Bodhi read daemon per node

Figure 6: Zazen cache read bandwidth on 100 nodes. (a) Forking one read daemon for each application read process hurts the performance significantly, especially when the size of files in the dataset is large. (b) We can eliminate the I/O contention by using a single Bodhi server read daemon per node to serialize the read requests.

algorithm.[1] As the number of nodes increases, the execution time grows only marginally, up to 14.9 seconds on 100 nodes.

The result is exactly as expected. When performing all-to-all reduction involving large messages, MPI libraries typically select a bandwidth-optimized ring algorithm [44], which we would have implemented had we not used MPI. The time required to execute the ring algorithm is $2(p-1)\alpha + 2n(1-1/p)\beta + n(1-1/p)\gamma$, where p is the number of processes, n is the size of the vector (i.e., the bitmap), α is the latency per message, β is the transfer time per byte, and γ is the computation cost per byte for performing the reduction operation. The coefficient associated with the bandwidth term, $2n(1-1/p)$, which is the dominant component for large messages, does not grow with the number of nodes (p).

Figure 5 shows that on 100 nodes, the execution time of the Zazen protocol grows sub-linearly as we increase the number of frames from 1,000 to 1,000,000,000. The result is again in line with the theoretical cost model of the ring algorithm, where the bandwidth term is linear in n, the size of the bitmaps.

To put the execution time of the Zazen protocol in perspective, let us assume that each frame of a simulation is 1 MB and we have one billion frames. The total size of such a dataset is one petabyte. Spending less than 15 seconds on 100 nodes to coordinate the parallel read of a petabyte-scale dataset appears (at least today) to be a reasonable startup overhead.

5.2 Efficiency

To measure the efficiency of actually reading data from the Zazen cache, we started the Bodhi servers on the 100 analysis nodes and populated the Zazen cache with four 1.6-TB test datasets, consisting of 1,600 1-GB files, 6,400 256-MB files, 25,600 64-MB files, and 819,200 2-MB files, respectively. Each node stored 16 GB of data on its local disks. The experiments were driven by an MPI program that executes the Zazen protocol and fetches the (whole) files in parallel from the local disks. No analysis was performed on the data and no cache misses occurred in these experiments.

In what follows, we report the end-to-end execution time measured between two `MPI_Barrier()` function calls placed before and after all Zazen cache operations. When reporting bandwidths, we compute them as the number of bytes read divided by the end-to-end execution time of reading the data. The numbers thus obtained are lower than the sum of locally computed I/O bandwidths since the slowest node would always drag down the overall bandwidth. Nevertheless, we choose to report the results in such an unfavorable way because it is a more realistic measurement of the actual I/O performance experienced by many analysis programs.

To ensure that the performance measurement was not aided in any way by the local file system buffer caches, we ran the experiments for reading the four datasets in a round-robin order and dropped the page, inode, and dentry caches from the Linux kernel before each individual experiment. We executed each experiment 5 times and computed the mean values. Because the coefficients of variation are negligible, we do not show error bars in the figures.

5.2.1 Effect of the Number of Bodhi Read Daemons

In this test, we compared the performance of two implementations of the Bodhi server to understand the effect of the number of read daemons. In the first implementation, we forked a new Bodhi server read process for each application read process and measured the performance of reading the four datasets on 100 nodes as shown in Figure 6(a). The dramatic drop between 1 and 2 readers per node for the 1-GB, 256-MB, and 64-MB datasets indicated that when two or more processes simultaneously read large data files, the interleaved I/O requests forced the disk sub-system to operate in a seek-bound mode, which significantly hurt the I/O performance. The further performance drop asso-

[1] Based on the vector size and the number of processes, Open MPI makes a runtime decision with respect to which all-reduce algorithm to use. The specifics are implementation dependent and are beyond the scope of this paper.

ciated with reading the 1-GB dataset using eight readers (and thus eight Bodhi read processes) per node was caused by double buffering: once within the application and once within the Bodhi read daemon. In total, 16 GB of memory—the total amount of physical memory on each node—was used for buffering the 1 GB files. As a result, the program suffered from memory thrashing and the performance plummeted. The degradation in performance associated with the 2-MB dataset was not as obvious since reading small files was already seek-bound even when only there is a single read process.

Based on this observation, we developed a second implementation of the Bodhi server and used a single Bodhi read daemon on each node to *serialize* all local client read requests. As a result, only one read request would be outstanding at any time while the rest would be waiting in a FIFO queue maintained internally by the Bodhi read daemon. Although serializing the parallel I/O requests may appear counterintuitive, Figure 6(b) shows that significantly better and more consistent performance across the spectrum was achieved.

5.2.2 Read-Only Performance

To compare the performance of Zazen with that of other representative systems, we measured the read-only I/O performance on NFS, a common, general-purpose network file system; PVFS, a widely deployed high-performance parallel file system [8, 31]; and Hadoop/HDFS [15], a popular, location-aware parallel file system. These experiments were set up as follows.

NFS. We used an enterprise NFS (v3.0) storage server with dual quad-core 2.8-GHz Opteron processors, 16 GB of memory, 48 SATA disks that are organized in RAID 6 and managed by ZFS, and four 1-GigE connections to the core switch of the 100-node analysis cluster. The total capacity of the NFS server is 40 TB. Anticipating lower read bandwidth (based on our prior experience), we generated four smaller test datasets consisting of 400 1-GB files, 400 256-MB files, 1,600 64-MB files, and 51,200 2-MB files, respectively, for the NFS experiments.

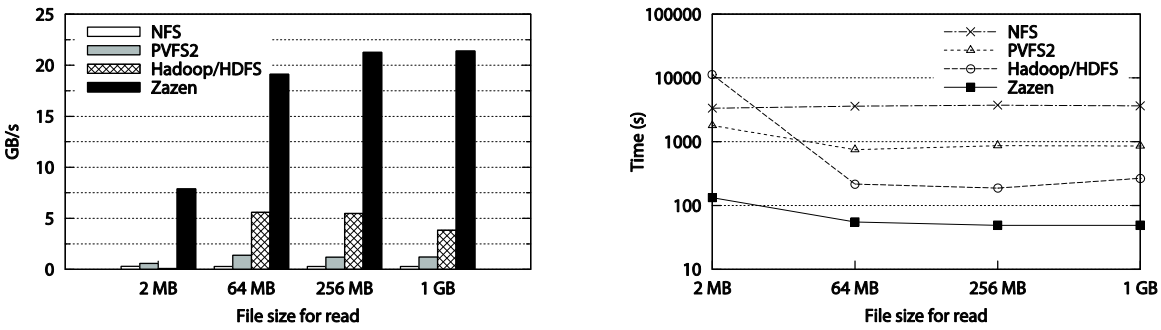
We modified the test program so that each process reads an equal number of data files from the mounted NFS directories. We ran the test program on 100 nodes and read the four datasets using 1, 2, and 4 cores per node, respectively. Seeing that the performance dropped consistently and significantly as we increased the number of cores per node, we did not run experiments using 8 cores per node. Each experiment (i.e., reading a dataset using a particular number of cores per node) was executed three times, all of which generated similar results (with negligible coefficients of variation). The highest performance was always obtained when one core per node was used to read the datasets, that is, when running 100 processes on 100 nodes. We report the best results from the one-core runs.

PVFS2. PVFS 2.8.1 was installed. All 100 analysis nodes ran both the I/O (data) server and the metadata server. The RAID 0 partitions on the analysis nodes were reformatted to provide the PVFS2 storage space. The PVFS2 Linux kernel interface was deployed and the PVFS2 volume was mounted locally on each node. The four datasets used to drive the evaluation of PVFS2 were the same as those used in the Zazen experiments. Data files were striped across all nodes.

The program used for driving the PVFS2 experiments was the same as the one used for the NFS experiments except that we pointed the data paths to the mounted PVFS2 directories. The PVFS2 experiments were conducted in the same way as the NFS experiments. The best results for reading the 1-GB and 256-MB datasets were attained with 2 cores per node, while the best results for reading the 64-MB and 2-MB datasets were obtained with 4 cores per node.

Hadoop/HDFS. Hadoop/HDFS release 0.19.1 was installed. We used the 100 analysis nodes as slaves (i.e., DataNodes and TaskTrackers) to store HDFS files and to execute MapReduce tasks. We also added three additional nodes to run the HDFS name node, the secondary name node, and the Hadoop MapReduce job tracker, respectively. We wrote and configured a rack awareness script for Hadoop/HDFS to identify the locations of the nodes.

The datasets we used to evaluate Hadoop/HDFS have the same characteristics as those for the Zazen and PVFS2 experiments. To store the datasets in HDFS efficiently, we wrote an MPI program that was linked with HDFS's C API library `libhdfs`. Considering that simulation analysis programs would process each frame as a whole (as a binary blob), we set the HDFS block size to be the same as the file size and did not split frame files across the slave nodes. Each file was replicated three times (the default setting) within HDFS. The data population program ran in parallel on 100 nodes and stored the data files uniformly on the 100 nodes.



(a) End-to-end read bandwidth comparison

(b) Time to read one terabyte data

Figure 7: Comparison of read-only performance. (a) Bars are grouped by the file size of the datasets, with the leftmost bar representing the performance of that of PVFS2, Hadoop/HDFS, and Zazen, respectively. (b) The y axis is shown in log-scale.

To read data efficiently from HDFS, we wrote a read-only Hadoop MapReduce program in Java. We used the following techniques to eliminate or minimize the overhead: (1) defining a `map()` function that returned immediately, so that no time would be spent in computation; (2) skipping the reduce phase, which was irrelevant for our experiments; (3) providing an unsplittable data input format to ensure that each frame file would be read as a whole on some node, and creating a binary record reader to read data in 64 MB chunks (when reading data files greater than or equal to 64 MB) so as to transfer data in bulk and avoid parsing overhead; (4) setting the output format to NULL type to avoid job output; (5) reusing the Java virtual machines for map task execution; and (6) setting the log file output to a local disk path on each node. In addition, we set the heap sizes for the name node and the job tracker to 8 GB and 15 GB, respectively, to allow maximum memory usage by Hadoop/HDFS.

Hadoop provides a configuration parameter to control the maximum number of map tasks that can be executed simultaneously on each slave node. We set this parameter to 1, 2, 4, 8, and 16, respectively, and executed the read-only MapReduce program to access the four test datasets. All experiments, except for those that read the 2-MB datasets, were performed three times, yielding similar results each time. We found that Hadoop had great difficulty in handling a large number of small files—a problem that had also been recognized by the Hadoop community [16]. The reading of the 2-MB dataset, which consisted of 819,200 files, failed multiple times when using a maximum of 1 or 2 map tasks per node, and took much longer than expected when 4, 8, and 16 map tasks per node were used. Hence, each experiment for reading the 2-MB dataset was performed only once. Regardless of the frame file size, setting the parameter to 8 led to the best results, which we use in the following performance comparison.

Figure 7(a) shows the read bandwidth delivered by the four systems. The bars are grouped by the file size of the datasets. Within each group, the leftmost bar represents the performance of NFS, followed by that of PVFS2, Hadoop/HDFS, and Zazen, respectively. Figure 7(b) shows the equivalent time (in log-scale) of reading 1 terabyte data of different file sizes. Zazen consistently outperforms other storage systems by a large margin across the range. When reading large files (i.e., 1-GB), Zazen delivers more than 20 GB/s sustained read bandwidth on the 100 nodes, outperforming NFS (on a single enterprise storage server) by a factor of 75, and PVFS2 and Hadoop/HDFS (running on the same 100 nodes) by factors of 18 and 6, respectively. When more seeks are required to read a large number of small (2-MB) files, Zazen achieves a sustained I/O bandwidth of about 8 GB/s, which is 25, 13, and 85 times faster than NFS, PVFS2, and Hadoop/HDFS, respectively. As a reference, the optimal aggregated disk read bandwidth we measured on the 100 nodes is around 22.5 GB/s. Zazen's I/O efficiency (up to 90%) is the direct result of "embarrassingly parallel" I/O operations that are enabled by the Zazen protocol.

We emphasize that despite Zazen's large performance advantage over file systems, it is intended to be used only as a disk cache to accelerate disk reads—just as processor caches are used to accelerate main memory accesses. Our results do not suggest that Zazen has the capability to replace the underlying file systems.

5.2.3 Read Performance under Write Workload

In this set of tests, we repeated the experiments of reading the four 1.6-TB datasets from the Zazen cache, while also concurrently executing Zazen cache writers. In particular, we used 8 additional nodes to act as supercomputer I/O nodes that continuously write to the 100-node Zazen cache at an aggregated rate of 1 GB/s.

Figure 8 shows the Zazen read performance under

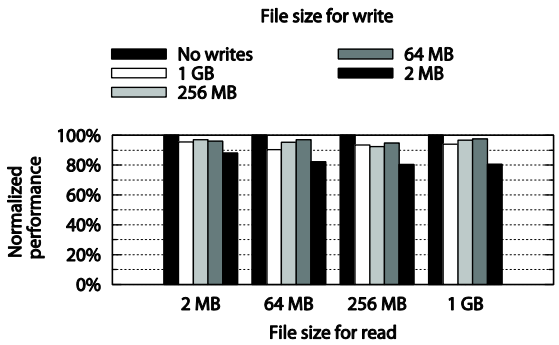


Figure 8: Zazen read performance under write workload. Writing data to the Zazen cache at a high rate (1 GB/s) does not affect the read performance in any significant way.

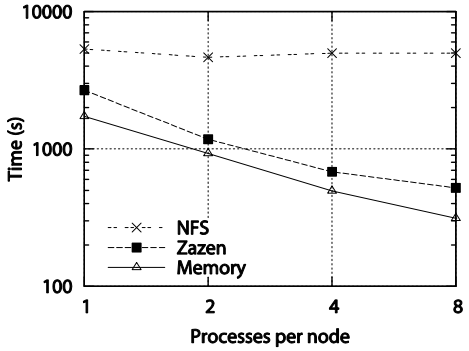


Figure 9: End-to-end execution time (100 nodes). Zazen enables the program to speed up as more cores per node are used.

write workload. The bars are grouped by the file size of the datasets being read. Within each group, the leftmost bar represents the read bandwidth attained with no writers, followed by the bars representing the read bandwidth attained while 1-GB, 256-MB, 64-MB, and 2-MB files are being written to the Zazen cache, respectively. The bars are normalized (divided) by the no-writer read bandwidth and shown as percentages.

We can see from the figure that Zazen achieves a high level of read performance (more than 90% of that obtained in the absence of writers) when medium to large files (64 MB–1 GB) were being written to the cache. Even in the most demanding case of writing 2-MB files, Zazen still delivers a performance above 80% of that measured in the no-writer case. These results demonstrate that actively pushing data into the Zazen cache does not significantly affect the read performance.

5.3 End-to-End Performance

We have deployed the 100-node Zazen cluster in conjunction with Anton and have used the cluster to execute hundreds of thousands of parallel analysis jobs. In general, we are able to reduce the end-to-end execution time of a large number of analysis programs—not just the data access time—from several hours to 5–15 minutes.

The sample application presented next is one of the most demanding in that it processes a large number (2.5 million) of small files (430-KB frames). The purpose of this analysis is to compute how long particular water molecules reside within a certain distance of a protein structure. The analysis program, called *water residence*, is a parallel Python program consisting of a data-extraction phase and a time-series analysis phase. I/O read takes place in the first phase when the frames are fetched and analyzed one file at a time (without a particular ordering requirement).

Figure 9 shows the performance of the sample program executing on the 100-node Zazen cluster. The three curves, from bottom up, represent the end-to-end execution time (in log-scale) when the program read data from (distributed) main memory, Zazen, and NFS, respectively. To obtain the reference time of reading frames directly from the main memory, we ran the program back-to-back three times without dropping the Linux cache in between so that the buffer cache of each of the 100 nodes is fully warmed. We used the measurement of the third run to represent the runtime for accessing data directly from main memory. Recall that the total memory of the Zazen cluster is 1.6 TB, which is sufficient to accommodate the entire dataset (1 TB). When reading data from the Zazen cache, we dropped the Linux cache before each experiment to eliminate any memory caching effect.

The memory curve represents the best possible scaling of the sample program, because no disk I/O is involved. As we increase the number of processes on each node, the execution time improves proportionally, because the same amount of computational workload is now split among more processor cores. The Zazen curve has a similar trend and closely follows the memory curve. The NFS curve, however, stays more or less flat regardless of how many cores are used on each node, from which we can see that I/O read is the dominant component of the total runtime, and that increasing the number of readers does not increase the effective I/O bandwidth. When we run eight user processes on each node, Zazen is able to improve the execution time of the sample program by 10 times over that attained by accessing data directly from NFS.

An attentive reader may recall from Figure 6(b) that increasing the number of application reader processes does not increase Zazen's read bandwidth either. Then why does the execution time when using the Zazen cache improve as we use more cores per node? The

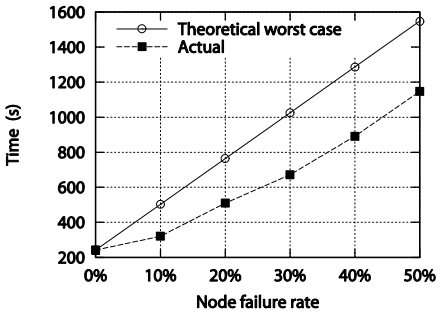


Figure 10: Performance under node failures. Individual node failures do not cause the Zazen system to crash.

reason is that the Zazen cache has reduced the I/O time to such an insignificant percentage of the application's total runtime that the computation time has now become the dominant component. Hence, doubling the number of cores per node not only halves the computation time, but also improves the overall execution time in a significant way. Another way to interpret the result is that by using the Zazen cache, we have turned an I/O-bound analysis into a computation-bound problem that is more amenable to parallel acceleration using multiple cores.

5.4 Robustness

Zazen is robust in that individual node crashes do not cause systemic failures. As explained in Section 4, the frame files cached on crashed nodes are simply treated as cache misses. To identify and exclude crashed or faulty nodes, we use a cluster resource manager called SLURM [39, 49] to schedule jobs and allocate nodes.

We assessed the effect of node failures on end-to-end performance by re-running the water residence program as follows. Before each experiment, we first purged the Zazen cache and then populated the 100 nodes with 1.25 million frame files uniformly. Next, we randomly selected a specified percentage of nodes and shut down the Bodhi servers on those nodes. Finally, we submitted the analysis job to SLURM, which detected the faulty nodes and excluded them from job execution.

Figure 10 shows the execution time of the water residence program along with the computed worst-case execution time as the percentage of failed nodes increases from 10% to 50%. The worst-case execution time can be shown to be $T(1 + \delta(B/b))$, where T is the execution time without node failures, δ is the percentage of the Zazen nodes that have failed, B is the aggregated I/O bandwidth of the Zazen cache without node failures, and b is the best read bandwidth of the underlying parallel/network file system. We measured, for this particular dataset, that B and b had values of 3.4 GB/s and 312 MB/s, respectively. Our results show that the actual execution time is indeed consistently below the computed worst-case time and degrades gracefully in the face of node failures.

6 Related Work

The idea of using local disks to accelerate I/O for scientific applications has been explored for over a decade. DPSS [45] is a parallel disk cache prototype designed to reduce I/O latency over the Grid. FreeLoader [47] aggregates the unused desktop disk space into a shared cache/scratch space to improve performance of single-client applications. Panache [1] uses GPFS [37] as a client-site disk cache and leverages the emerging parallel NFS standard [29] to improve cross-WAN data access performance. Zazen shares the philosophy of these systems but has a different goal: it aims to obtain the best possible aggregated read bandwidth from local cache nodes rather than reducing remote I/O latency.

Zazen does not attempt to provide a location-transparent view of the cached data to applications. Instead of confederating a set of distributed disks into a single, unified data store—as do the distributed/parallel disk cache systems and cluster file systems such as PVFS [8], Lustre [21], and GFS [13]—Zazen converts distributed disks into a collection of independently managed caches that are accessed in parallel by a large number of cooperative application processes.

While existing works such as Active Data Repository [19] uses spatial index structures (e.g., R-trees) to select a subset of a multidimensional dataset and thus effectively reduces I/O workload and enables interactive visualization, Zazen targets a simple data access pattern of one-frame-at-a-time and strives to improve the I/O performance of batch analysis.

Peer-to-peer (P2P) storage systems, such as PAST [34], CFS [9], Ivy [24], Pond [32], and Kosha [7], also do not use centralized or dedicated servers to keep track of distributed data. They employ a scalable technique called a *distributed hash table* [2] to route lookup requests through an overlay network to a peer where the data are stored. These systems differ from Zazen in three essential ways. First, P2P systems target completely decentralized and largely unrelated machines, whereas Zazen attempts to harness the power of tightly coupled cluster nodes. Second, while P2P systems use distributed coordination to provide high availability, Zazen relies on global coordination to achieve consensus and thus high performance. Third, P2P systems, as the name suggests, send and receive data over the network among peers. In contrast, Zazen accesses data *in situ* whenever possible; data traverse the network only when a cache miss occurs.

Although similar in spirit to GFS/MapReduce [10, 13], Hadoop/HDFS [15], Gfarm [41, 42], and Tashi [18], all of which seek data location information from metadata servers to accelerate parallel processing

of massive data, Zazen employs an unorthodox approach to identify the whereabouts of the stored data, and thus avoids the potential performance and scalability bottleneck and the single point of failure associated with metadata servers.

At the implementation level, Zazen caches whole files like AFS [17, 35] and Coda [36], though bookkeeping in Zazen is much simpler as simulation output files are immutable and do not require leases and callbacks to maintain consistency. The use of bitmaps in the Zazen protocol bears resemblance to the version vector technique [27] used in the LOCUS system [48]. While the latter associated a version vector with each copy of a file to detect and resolve conflicts among distributed replicas, Zazen uses a more compact representation to arbitrate who should read which frame files.

7 Summary

As parallel scientific supercomputing enters a new era of scale and performance, the pressure on post-simulation data analysis has mounted to such a point that a new class of hardware/software systems has been called for to tackle the unprecedented data problems [3]. The Zazen system presented in this paper is the storage subsystem underlying a large analysis framework that we have been developing.

With the intention to deploy Zazen to cache millions to billions of frame files and execute on hundreds to thousands of processor cores, we conceived a new approach by exploiting the characteristics of a particular class of time-dependent simulation datasets. The outcome was an implementation that delivered an order-of-magnitude end-to-end speedup for a large number of parallel trajectory analysis programs.

While our work was motivated by the need to accelerate parallel analysis programs that operate on very long trajectories consisting of relatively small frames, we envision that the method, techniques, and algorithms described here can be adapted to support other kinds of data-intensive parallel applications. In particular, if the data objects of an application can be interpreted as having a total ordering of some sort (e.g. in the temporal or spatial domain), then unique sequence numbers can be assigned to identify the data objects. These datasets would appear no different from time-dependent scientific simulation datasets and thus would be amenable to I/O acceleration via Zazen.

References

[1] R. Ananthanarayanan, M. Eshel, R. Haskin, M. Naik, F. Schmuck, and R. Tewari. Panache: a parallel WAN cache for clustered filesystems. *ACM SIGOPS Operating Systems Review*, 42(1):48–53, January 2008.

[2] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking up data in P2P systems. *Communications of the ACM*, 46(2):43–48, February 2003.

[3] G. Bell, T. Hey, and A. Szaley. Beyond the data deluge. *Science*, 323(5919):1297–1298, March 2009.

[4] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, et al. PLFS: a checkpoint filesystem for parallel applications. In *Proceedings of the 2009 ACM/IEEE Conference on Supercomputing (SC09)*, Portland, OR, November 2009.

[5] J. Bent, D. Thain, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. Explicit control in a batch-aware distributed file system. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI'04)*, San Francisco, CA, March 2004.

[6] J. Bruck, C-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1143–1156, November 1997.

[7] A. R. Butt, T. A. Johnson, Y. Zheng, and Y. C. Yu. Kosha: a peer-to-peer enhancement for the network file system. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC04)*, Pittsburgh, PA, November 2004.

[8] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. PVFS: a parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, October 2000.

[9] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, pages 202–215, Banff, Alberta, Canada, October 2001.

[10] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, January 2008.

[11] G. E. Fagg, J. Pjesivac-Grbovic, G. Bosilca, T. Angskun, J. J. Dongarra, and E. Jeannot. Flexible collective communication tuning architecture applied to Open MPI. In *Proceedings of the 13th European PVM/MPI Users' Group Meeting (Euro PVM/MPI 2006)*, Bonn, Germany, September 2006.

[12] I. Foster, D. Kohr, R. Krishnaiyer, and J. Mogill. Remote I/O: fast access to distant storage. In *Proceedings of the 5th Workshop on Input/Output in Parallel and Distributed Systems*, pages 14–25, San Jose, CA, November 1997.

[13] S. Ghemawat, H. Gobioff, and S-T. Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, Bolton Landing, NY, October 2003.

[14] GridFTP. `http://www.globus.org/grid_software/data/gridftp.php/`.

[15] Hadoop. `http://hadoop.apache.org/`.

[16] Hadoop/HDFS small files problem. `http://www.cloudera.com/blog/2009/02/02/the-small-files-problem/`.

[17] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, et al. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[18] M. A. Kozuch, M. P. Ryan, R. Gass, S. W. Schlosser, D. R. O'Hallaron, et al. Tashi: location-aware cluster management. In *Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds (ACDC09)*, Barcelona, Spain, June 2009.

[19] T. Kurc, Ü Çatalyürek, C. Chang, A. Sussman, and J. Saltz. Visualization of Large Data Sets with the Active Data Repository. IEEE Computer Graphics and Applications, 21(4):24–33, July/August 2001.

[20] J. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin. Flexible IO and Integration for Scientific Codes Through The

Adaptable IO System (ADIOS). In *Proceedings of the 6th ACM/IEEE International Workshop on Challenges of Large Applications in Distributed Environments* (CLADE.2008), Boston, MA, June 2008.

[21] Lustre. http://www.sun.com/software/products/lustre/.

[22] H. M. Monti, A. R. Butt, and S. S. Vazhkudai. Just-in-time staging of large input data for supercomputing jobs. In *Proceedings of the 3rd Petascale Data Storage Workshop*, Austin, TX, November 2008.

[23] H. M. Monti, A. R. Butt, and S. S. Vazhkudai. /scratch as a cache: rethinking HPC center scratch storage. In *Proceedings of the 23rd International Conference on Supercomputing*, Yorktown Height, NY, June 2009.

[24] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: a read/write peer-to-peer file system. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*, Boston, MA, November 2002.

[25] P. Nowoczynski, N. Stone, J. Yanovich, and J. Sommerfield. Zest: checkpoint storage system for large supercomputers. In *Proceedings of the 3rd Petascale Data Storage Workshop*, Austin, TX, November 2008.

[26] Open MPI. http://www.open-mpi.org/.

[27] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, et al. Detection of mutual inconsistency in distributed systems. *IEEE Trascations on Software Engineering*, 9(3):240–247, May 1983.

[28] Petascale Data Storage Institute. http://www.pdsi-scidac.org/.

[29] Parallel NFS. http://www.pnfs.com/.

[30] D. H. Porter, P. R. Woodward, and A. Iyer. Initial experiences with grid-based volume visualization of fluid flow simulations on PC clusters. In *Proceedings of Visualization and Data Analysis 2005 (VDA2005)*, San Jose, CA, January 2005.

[31] PVFS. http://www.pvfs.org/.

[32] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the OceanStore prototype. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST'03)*, San Francisco, CA, March 2003.

[33] ROMIO. http://www.mcs.anl.gov/research/projects/romio/

[34] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, Banff, Alberta, Canada, November 2001.

[35] M. Satyanarayanan, J. H. Howard, D. A. Nichols, R. N. Sidebotham, A. Z. Spector, and M. J.West. The ITC distributed file system: principles and design. In *Proceedings of the 10th ACM symposium on Operating Systems Principles*, Orcas Island, WA, 1985.

[36] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: a highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, April 1990.

[37] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST'02)*, Monterey, CA, January 2002.

[38] D. E. Shaw, R. O. Dror, J. K. Salmon, J. P. Grossman, K. M. Mackenzie, et al. Millisecond-scale molecular dynamics simulation on Anton. In *Proceedings of the 2009 ACM/IEEE Conference on Supercomputing (SC09)*, Portland, OR, November 2009.

[39] SLURM. https://computing.llnl.gov/linux/slurm/slurm.html/.

[40] N. T. B. Stone, D. Balog, B. Gill, B. Johanson, J. Marsteller, et al. PDIO: high-performance remote file I/O for Portals enabled compute nodes. In *Proceedings of the 2006 Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, NV, June 2006.

[41] O. Tatebe, Y. Morita, S. Matsuoka, N. Soda, and S. Sekiguchi. Grid datafarm architecture for petascale data intensive computing. In *Proceedings of the 2nd IEEE/ACM Internaiontal Symposium on Cluster Computing and the Grid (CCGrid2002)*, Berlin, Germany, May 2002.

[42] O. Tatebe, N. Soda, Y. Morita, S. Matsuoka, and S. Sekiguchi. Gfarm v2: A grid file system that supports high-performance distributed and parallel data computing. In *Proceedings of the 2004 Computing in High Energy and Nuclear Physics*, Interlaken, Switzerland, September 2004.

[43] R. Thakur, W. Gropp, and E. Lusk. An abstract-device interface for implementing portable parallel-I/O interfaces. In *Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation*, Annapolis, MD, October 1996.

[44] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in MPICH. *International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[45] B. L. Tierney, J. Lee, B. Crowley, M. Holding, J. Hylton, and F. L. Drake Jr. A network-aware distributed storage cache for data intensive environments. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC-8)*, Redondo Beach, CA, August 1999.

[46] T. Tu, C. A. Rendleman, D. W. Borhani, R. O. Dror, J. Gullingsrud, et al. A Scalable Parallel Framework for Analyzing Terascale Molecular Dynamics Simulation Trajectories. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC08)*, Austin, Texas, November 15–21, 2008.

[47] S. S. Vazhkudai, X. Ma, V.W. Freeh, J.W. Strickland, N. Tammineedi, and S. L. Scott. FreeLoader: scavenging desktop storage resources for scientific data. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (SC05)*, Settle, WA, November 2005.

[48] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS distributed operating system. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, Bretton Woods, MA, October 1983.

[49] A. Yoo, M. Jette, and M. Grondona. SLURM: simple Linux utility for resource management. In *Lecture Notes in Computer Science*, volume 2862 of *Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer Berlin/Heidelberg, 2003.

Efficient Object Storage Journaling in a Distributed Parallel File System

Sarp Oral, Feiyi Wang, David Dillow, Galen Shipman, Ross Miller
National Center for Computational Sciences
Oak Ridge National Laboratory
{*oralhs,fwang2,gshipman,dillowda,rgmiller*}*@ornl.gov*

Oleg Drokin
Lustre Center of Excellence at ORNL
Sun Microsystem Inc.
oleg.drokin@sun.com

Abstract

Journaling is a widely used technique to increase file system robustness against metadata and/or data corruptions. While the overhead of journaling can be masked by the page cache for small-scale, local file systems, we found that Lustre's use of journaling for the object store significantly impacted the overall performance of our large-scale center-wide parallel file system. By requiring that each write request wait for a journal transaction to commit, Lustre introduced serialization to the client request stream and imposed additional latency due to disk head movement (seeks) for each request.

In this paper, we present the challenges we faced while deploying a very large scale production storage system. Our work provides a head-to-head comparison of two significantly different approaches to increasing the overall efficiency of the Lustre file system. First, we present a hardware solution using external journaling devices to eliminate the latencies incurred by the extra disk head seeks due to journaling. Second, we introduce a software-based optimization to remove the synchronous commit for each write request, side-stepping additional latency and amortizing the journal seeks across a much larger number of requests.

Both solutions have been implemented and experimentally tested on our Spider storage system, a very large scale Lustre deployment. Our tests show both methods considerably improve the write performance, in some cases up to 93%. Testing with a real-world scientific application showed a 37% decrease in the number journal updates, each with an associated seek – which translated into an average I/O bandwidth improvement of 56.3%.

1 Introduction

Large-scale HPC systems target a balance of file I/O performance with computational capability. Traditionally, the standard was 2 bytes per second of I/O bandwidth for each 1,000 FLOPs of computational capacity [18]. Maintaining that balance for a 1 Petaflops (PFLOPs) supercomputer would require the deployment a storage subsystem capable of delivering 2 TB/sec of I/O bandwidth at a minimum. Building such a system with current or near-term storage technology would require on the order of 100,000 magnetic disks. This would be cost prohibitive not only due to the raw material costs of the disks themselves, but also to the magnitude of the design, installation, and ongoing management and electrical costs for the entire system, including the RAID controllers, network links, and switches. At this scale, reliability metrics for each component would virtually guarantee that such a system would continuously operate in a degraded mode due to ongoing simultaneous reconstruction operations [22].

The National Center for Computational Sciences (NCCS) at Oak Ridge National Laboratory (ORNL) hosts the world's fastest supercomputer, Jaguar [8] with over 300 TB of total system memory. Rather than rely on a traditional I/O performance metric such as 2 byte/sec of I/O throughput for each 1000 FLOP of computational capacity a survey of application requirements was conducted prior to the design of the parallel I/O environment for Jaguar. This resulted in a requirement of delivered bandwidth of over 166 GB/sec based on the ability to checkpoint 20% of total system memory, once per hour, using no more than 10% of total compute time. Based on application I/O profiles and available resources, the Jaguar upgrade targeted 240 GB/s of storage bandwidth. Achieving this target on Jaguar has required a careful attention to detail and optimization of the

system at multiple levels, including the storage hardware, network topology, OS, I/O middleware, and application I/O architecture.

There are many studies on user-level file system performance of different Cray XT platforms and their respective storage subsystems. These provide important information for scientific application developers and system engineers such as peak system throughput and the impact of Lustre file striping patterns [33, 1, 32]. However – to the best of our knowledge – there has been no work done to analyze the efficiency of the object storage system's journaling and its impact of overall I/O throughput in a large-scale parallel file system such as Lustre.

Journaling is widely used by modern file systems to increase file system robustness against metadata corruptions and to minimize file system recovery times after a system crash. Aside from journaling, there are several other techniques for preventing metadata corruption. Soft updates handle the metadata update problem by guaranteeing that blocks are written to disk in their required order without using synchronous disk I/O [10, 23]. Vendors such as Network Appliance [3], have addressed the issue with a hardware assisted approach (non-volatile RAM) resulting in performance superior to both journaling and soft updates at the expense of extra hardware. NFS version 3 [20] introduced asynchronous writes to overcome the bottleneck of synchronous writes. The server is permitted to reply to the client before the data is on stable storage, which is similar to our Lustre asynchronous solution. The Log-based file system [17] took a departure from the conventional update-in-place approach by writing modified data and metadata in a log. More recently, ZFS [13] has been coupled with flash-based devices for intent logging so that synchronous writes are directed to these log devices with very low latency, improving overall performance.

While the overhead of journaling can be masked by using the page cache for local file systems, our experiments show that on a large-scale parallel Lustre file system it can substantially degrade overall performance.

In this paper, we present our experiences and the challenges we faced towards deploying a very large scale production storage system. Our findings suggest that sub-optimal object storage file system journaling performance significantly hurts the overall parallel file system performance. Our work provides a head-to-head comparison of two significantly different approaches to increasing overall efficiency of the Lustre file system. First, we present a hardware solution using external journaling devices to eliminate the latencies incurred by the extra disk head seeks for the journal traffic. Second, we introduce a software-based optimization to remove the synchronous commit for each write request, side-stepping additional latency and amortizing the journal seeks across a much larger number of requests.

Major contributions of our work include measurements and performance characterization of a very large storage system unique in its scale; The identification and elimination of serial bottlenecks in a large-scale parallel system; A cost-effective and novel solution to file system journaling overheads in a large scale system.

The remainder of this paper is organized as follows: Section 2 introduces Jaguar and its large-scale parallel I/O subsystem, while Section 3 provides a quick overview of the Lustre parallel file system and presents our initial findings on the performance problems Lustre file system journaling. Section 4 introduces our hardware solution to the problem and Section 5 presents the software solution. Section 6 summarizes and provides a discussion on results of our hardware and software solutions and presents results of real science application using our software-based solution. Section 7 presents our conclusions.

2 System Architecture

Jaguar is the main simulation platform deployed at ORNL. Jaguar entered service in 2005 and has undergone several upgrades and additions since that time. Detailed descriptions and performance evaluations of earlier Jaguar iterations can be found in the literature [1].

2.1 Overview of Jaguar

In late 2008, Jaguar was expanded with the addition of a 1.4 PFLOPs Cray XT5 in addition to the existing Cray XT4 segment[1]. Resulting in a system with over 181,000 processing cores connected internally via Cray's SeaStar2+ [4] network. The XT4 and XT5 segments of Jaguar are connected via a DDR InfiniBand network that also provides a link to our center-wide file system, Spider. More information about the Cray XT5 architecture and Jaguar can be found in [5, 19].

Jaguar has 200 Cray XT5 cabinets. Each cabinet has 24 compute blades. Each blade has 4 compute nodes and each compute node has two AMD Opteron 2356 Barcelona quad-core processors. Figure 1 shows the high-level Cray XT5 node architecture. The configuration tested, has 16 GB of DDR2-800 MHz memory per compute node (2 GB per core), for a total of 300 TB of system memory. Each processor is linked with dual HyperTransport connections. The HyperTransport interface enables direct high-bandwidth connections between the processor, memory and the SeaStar2+ chip. The result is a dual-socket, eight-core node with a peak processing performance of 73.6 GFLOPS.

[1] A more recent Jaguar XT5 upgrade swapped the quad-core AMD Opteron 2356 CPUs (Barcelona) with hex-core AMD Opteron 2435 CPUs (Istanbul), increasing the installed peak performance of Jaguar XT5 to 2.33 PFLOP and total number of cores to 224,256.

The XT5 segment has 214 service and I/O nodes, of which 192 provide connectivity to the Spider center-wide file system with 240 GB/s of demonstrated file system bandwidth over the scalable I/O network (SION). SION is deployed as a multi-stage InfiniBand network [25], and provides a backplane for the integration of multiple NCCS systems such as Jaguar (the simulation and analysis platform), Spider (the NCCS-wide Lustre file system), Smoky (the development platform), and various other compute resources. SION allows capabilities such as streaming data from the simulation platforms to the visualization center at extremely high rates.

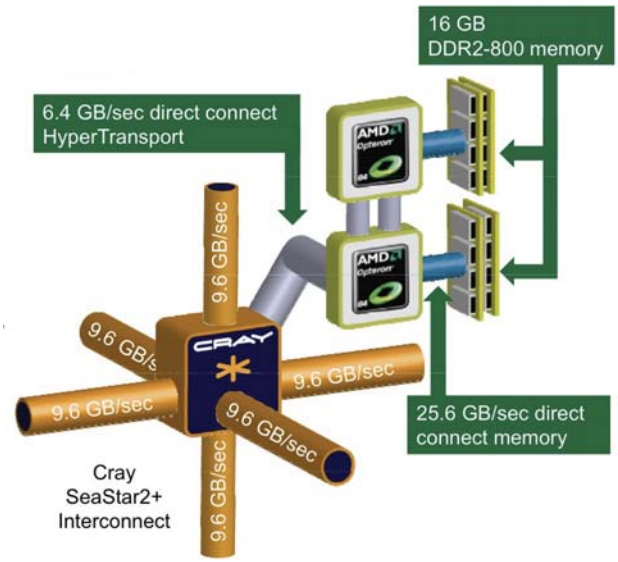


Figure 1. Cray XT5 node (*courtesy of Cray*)

2.2 Spider I/O subsystem

The Spider I/O subsystem consists of Data Direct Networks' (DDN) S2A9900 storage devices interconnected via SION. A pair of S2A9900 RAID controllers is called a couplet. Each controller in a couplet works as an active-active fail-over unit. There are 48 DDN S2A9900 couplets [6] in the Spider I/O subsystem. Each couplet is configured with five ultra-high density 4U, 60-bay disk drive enclosures (56 drives populated), giving a total of 280 1TB hard drives per S2A9900. The system as whole has 13,440 TB or over 10.7 PB of formatted capacity. Fig. 2 illustrates the internal architecture of a DDN S2A9900 couplet. Two parity drives are dedicated in the case of an 8+2 parity group or RAID 6. A parity group is also known as a **Tier**.

Spider, the center-wide Lustre [28] file system, is built upon this I/O subsystem. Spider is the world's fastest and largest production Lustre file system and is one of the

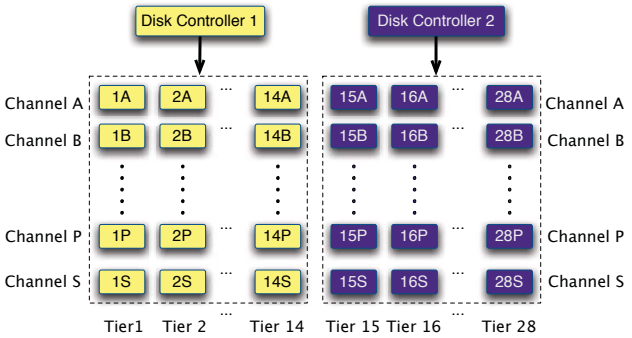


Figure 2. Architecture of a S2A9900 couplet

world's largest POSIX-compliant file systems. It is designed to work with both Jaguar and other computing resources such as the visualization and end-to-end analysis clusters. Spider has 192 Dell PowerEdge 1950 servers [7] configured as Lustre servers presenting a global file system name space. Each server has 16 GB of memory and dual socket, quad core Intel Xeon E5410 CPUs running at 2.3 GHz. Each server is connected to SION and the DDN arrays via independent 4x DDR InfiniBand links. In aggregate, Spider delivers up to 240 GB/s of file system level throughput and provides 10.7 PB of formatted disk capacity to it users. Fig. 3 shows the overall Spider architecture. More details on Spider can be found in [26].

3 Lustre and file system journaling

Lustre is an open-source distributed parallel file system developed and maintained by Sun Microsystems and licensed under the GNU General Public License (GPL). Due to the extremely scalable architecture of Lustre, deployments are popular in both scientific supercomputing and industry. As of June 2009, 70% of the Top 10 systems, 70% of the Top 20 and 62% of the Top 50 fastest supercomputers systems in the world used Lustre for high-performance scratch space [9], including Jaguar[2].

3.1 Lustre parallel file system

Lustre is an object-based file system and is composed of three components: Metadata storage, object storage, and clients. There is a single metadata target (MDT) per file system. A metadata server (MDS) is responsible for managing one or more MDTs. Each MDT stores file metadata, such as file names, directory structures, and access permissions. Each object storage server (OSS) manages one or more object storage targets (OSTs) and OSTs store file data objects.

[2] As of November 2009, 60% of the Top 10 fastest supercomputers systems in the world used Lustre file system for high-performance scratch space, including Jaguar.

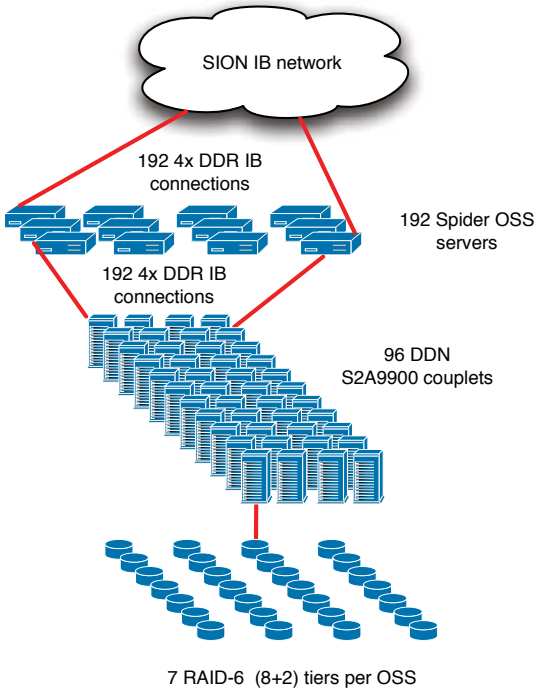


Figure 3. Overall Spider architecture

For file data read/write access, the MDS is not on the critical path, as clients send requests directly to the OSSes. Lustre uses block devices for file data and metadata storage and each block device can only be managed by one Lustre service (such as an MDT or an OST). The total data capacity of a Lustre file system is the sum of all individual OST capacities. Lustre clients concurrently access and use data through the standard POSIX I/O system calls. More details on the inner workings of Lustre can be found in [31].

Currently, Lustre version 1.6 employs a heavily patched and enhanced version of the Linux *ext3* file system, known as *ldiskfs*, as the back-end local file system for the MDT and all OSTs. Among the enhancements, improvements over the regular *ext3* file system journaling are of particular interest for this paper and will be covered in the next sections.

3.2 File system journaling in Lustre

A journaling file system, such as *ext3*, keeps a log of metadata and/or file data updates and changes so that in case of a system crash, file system consistency can be restored quickly and easily [30]. The file system can journal only the metadata updates or both metadata and data updates, depending on the implementation. The design choice is to balance file system consistency requirements against performance penalties due to extra journaling write operations and delays. In Linux *ext3*, there are three different modes of journaling: *write-back mode*, *ordered mode*, and *data journaling mode*. In *write-back mode*, updated metadata blocks are written to the journal device while file data blocks are written directly to the block device. When the transaction is committed, journaled metadata blocks are flushed to the block device without any ordering between the two events. *Write-back mode* thus provides metadata consistency but does not provide any file data consistency. In *ordered mode*, file data is guaranteed to be written to their fixed locations on disk before committing the metadata updates to the journal. This ordering protects the metadata and prevents stale data from appearing in a file in the event of a crash. *Data journaling mode* journals both the metadata and the file data. More details on *ext3* journaling modes and their performance characteristics can be found in [21].

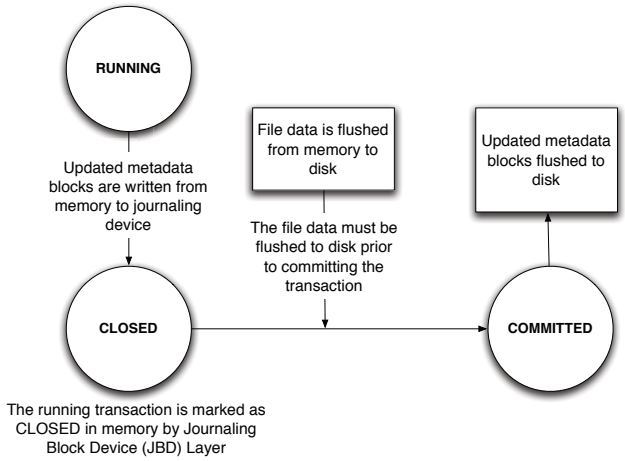


Figure 4. Flow diagram for the ordered mode journaling.

Although in the latest Linux kernels the default journaling mode for *ext3* file system is a build-time kernel configuration switch (between *ordered mode* and *write-back mode*), *ordered mode* is the default journaling mode for the *ldiskfs* file system used as the object store in Lustre.

Journaling in *ext3* is organized such that at any given time there are two transactions in memory (not written to the journaling device yet): the *currently running transaction* and the *currently closed transaction* (that is being committed to the disk). The currently running transaction is open and accepting new threads to join in and has all its data still in memory. The currently closed transaction is not accepting any new threads to join in and has started flushing its updated metadata blocks from memory to the journaling device. After the flush operation is complete and all transactions are on stable storage, the transaction state will be changed to "*committed*." The currently running transaction

can not be closed and committed until the closed transaction fully commits to the journaling device, which for slow disk subsystems can be a point of serialization. Also, even when the disk subsystem is relatively fast, there is another potential point of serialization due to the size of the journaling device. The largest transaction size that can be journaled is limited to 25% of the size of the journal. When a transaction reaches the limit, it is locked and will not accept any new threads or data.

The following list summarizes the steps taken by *ldiskfs* for a Lustre file update in the default ordered journaling mode. The sequence of events is triggered by a Lustre client sending a write request to an OST.

1. Server gets the destination object id and offset for this write operation.
2. Server allocates necessary number of pages in memory and fetches the data from the remote client into the these pages via an Remote Memory Access (RMA) GET operation.
3. Server opens a transaction on its back-end file system.
4. Server updates file metadata in memory, allocates blocks and extends the file size.
5. Server closes transaction handle and obtains a wait handle, but does **not** commit to journaling device.
6. Server writes pages with file data to disk synchronously.
7. After current running transaction is closed, server flushes updated metadata blocks to the journal device and then marks the transaction as committed.
8. Once transaction is committed, server can send a reply to client that the operation was completed successfully and client marks the request as completed.

   Also, the updated metadata blocks, which have been committed to journal device by now will be written to disk, without particular ordering requirement. Fig. 4 shows the generic outline of ordered mode journaling.

There is a minor difference between how this sequence of events happen on an *ext3* file system and the Lustre *ldiskfs* file system. In an *ext3* file system the sequence of steps 6 and 7 are strictly preserved. However, in Lustre *ldiskfs*, the metadata commit can happen before all data from Step 6 is on disk, Step 7 (flushing of updated metadata blocks to the journaling device) can partially happen before Step 6.

Although Step 5 minimizes the time a transaction is kept open, the above sequence of events may be sub-optimal. For example:

- An extra disk head seek is needed for the journal transaction commit after flushing file data on a different sector of the disk if the journaling device is located on the same device as the block file data.
- The write I/O operation for a new thread is blocked on the currently closed transaction which is committing on Step 7.
- The new running journal transaction has to wait for the previous transaction to be closed.
- New I/O RPCs are not formed until the completion replies of the previous RPCs have been received by the client creating yet another point of serialization.

The *ldiskfs* file system by default performs journaling in *ordered mode* by first writing the data blocks to disk followed by metadata blocks to the journal. The journal is then written to disk and marked as committed. In the worst case, such as appending to a file, this can result in one 16 KB write (on average – for bitmap, inode block map, inode, and super block data) and another 4 KB write for the journal commit record for every 1 MB write. These extra small writes cause at least two extra disk head seeks. Due to the poor IOP performance of SATA disks, these additional head seeks and small writes can substantially degrade the aggregate block I/O performance.

A potential optimization (and perhaps the most obvious one) for *ordered mode* to improve the journaling efficiency is to minimize the extra disk head seeks. This can be achieved by either a software or hardware optimization (or both). Section 4 describes our hardware based optimization while Section 5 discusses our software based optimization.

Using journaling methods other than *ordered mode* (or no journaling at all) in the *ldiskfs* file system is not considered in this study, as the OST handler waits for the data writes to hit the disk before returning, and only the metadata is updated in an asynchronous manner. Therefore, *write-back mode* would not help in our case – Lustre would not use the write-back functionality. *Data journaling mode* provides increased consistency and satisfies the Lustre requirements, but we would expect it to result in a reduction of performance from our pre-optimization baseline due to doubling the amount of bulk data written. Of course, running without any journaling is a possibility for obtaining better performance, but the cost of possible file system inconsistencies in a production environment is a price that we could ill afford.

To better understand the performance characteristics of each implementation we have performed a series of tests to obtain a baseline performance of our configuration. In order to obtain this baseline on the DDN S2A9900, the XDD benchmark [11] utility was used. XDD allows multiple clients to exercise a parallel write or read operation

synchronously. XDD can be run in sequential or random read or write mode. Our baseline tests focused on aggregate performance for sequential read or write workloads. Performance results using XDD from 4 hosts connected to the DDN via DDR IB are summarized in Fig. 1. The results presented are a summary of our testing and show performance of sequential read, sequential write, random read, and random write using 1MB transfers. These tests were run using a single host for the single LUN tests, and 4 hosts each with 7 LUNs for the 28 LUN test. Performance results presented are the best of 5 runs in each configuration.

Table 1. XDD baseline performance

		Read	Write
Single LUN	Sequential	685.62	235.45
	Random	101.74	96.77
28 LUN	Sequential	5570.15	5608.15
	Random	2753.87	2530.5

After establishing a baseline of performance using XDD, we examined Lustre level performance using the IOR benchmark [24]. Testing was conducted using 4 OSSs each with 7 OSTs on the DDN S2A9900. Our initial results showed very poor write performance of only 1,398.99 MB/sec using 28 clients where each client was writing to different OST. Lustre level write performance was a mere 24.9% of our baseline performance metric of XDD sequential writes with a 1MB transfer size. Profiling the I/O stream of the IOR benchmark using the DDN S2A9900 utilities revealed a large number of 4 KB writes in addition to the expected 1 MB writes. These small writes were traced to *ldiskfs* journal updates.

4 The Hardware Solution

To separate small-sized metadata journal updates from larger (1 MB) block I/O requests and thus enhance our aggregate block I/O performance, we evaluated two hardware-based solutions. Our first option was to use SAS drives as external journal devices. SAS drives are proven to have higher IOP performance compared to SATA drives. For this purpose we used two tiers of SAS drives in a DDN S2A9900, and each tier was split into 14 LUNs. Our second option was to use an external solid state device as the external journaling device. Although the best solution is to provide a separate disk for journaling for each file block device (or even a tier of disks as a single journaling device for each file block device tier), this is highly cost prohibitive at the scale of Spider.

Unlike rotating magnetic disks, solid state disks (SSD) have a negligible seek penalty. This makes SSDs an attractive solution for latency-sensitive storage workloads. SSDs can be flash memory based or DRAM or SRAM based. Furthermore, in recent years, solid state disks have become much more reasonable in terms of cost per GB [14]. The nearly zero seek latency of SSDs make them a logical choice to alleviate our Lustre journaling performance bottleneck.

We have evaluated Texas Memory Systems' RamSan-400 device [29] (on loan from the ViON Corp.) to assess the efficiency of an SSD based Lustre journaling solution for the Spider parallel file system. The RamSan is a 3U rackable solution and has been optimized for high transactional aggregate performance (400,000 small I/O operations per second). The RamSan-400 is a non-volatile SSD with backup hard drives configured as a RAID-3 set. The front end non-volatile solid state disks are a proprietary implementation of Texas Memory Systems' using highly redundant DDR RAM chips. The RamSan-400's block I/O performance is advertised by the vendor at an aggregate of 3 GB/sec. It is equipped with four 4x DDR InfiniBand host ports and supports the SCSI RDMA protocol (SRP).

For our testing purposes, we have connected the RamSan device to our SION network via four 4x DDR IB links directly to the Core 1 switch. This configuration allowed the Lustre servers (MDS and OSSes) to have direct connections to the LUNs on the RamSan device. We configured 28 LUNs (one for each Lustre OST, 7 per each IB host port) on the RamSan device. Fig. 5 presents our experiment layout.

Each LUN on the RamSan was formatted as an external *ldiskfs* journal device and we established a one-to-one mapping between the external journal devices and the 28 OST block devices on one DDN S2A9900 RAID controller. The *obdfilter-survey* benchmark [27] was used for testing both the SAS disk-based and the RamSan-based solutions. *Obdfilter-survey* is part of the Lustre I/O kit and it allows one to exercise the underlying Lustre file system with sequential I/O with varying numbers of threads and objects (files). *Obdfilter* can be used to characterize the performance of the Lustre network, individual OSTs, and the striped file system performance (including multiple OSTs and the Lustre network components). For more details on *obdfilter* readers are encouraged to read the Lustre User Manual [28]. Fig. 6 presents our results for these tests.

For comparative analysis, we ran the same *obdfilter-survey* benchmark on three different target configurations. The first target had external journals on a tier of SAS drives in the DDN S2A900, the second target had external journals on the RamSan-400 device, and third target had internal journals on a tier of SATA drives on our DDN S2A900. We varied the number of threads for each target while measuring the observed block I/O bandwidth. Both solutions with external journals provided good performance improvements. Internal journals on the SATA drives performed the

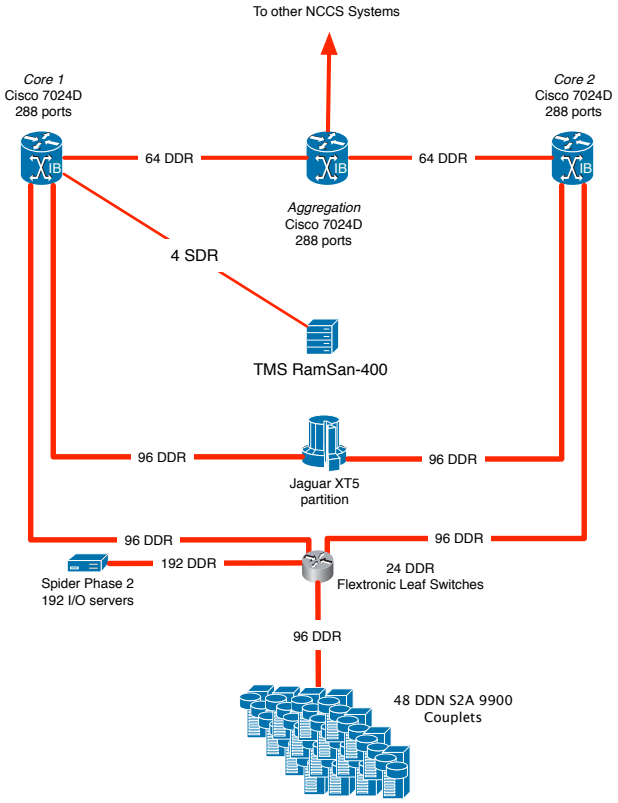


Figure 5. Layout for Lustre external journaling experiment with a RamSan-400 solid state device. The RamSan-400 was connected to the SION network via 4 DDR links and each link exported 7 LUNs.

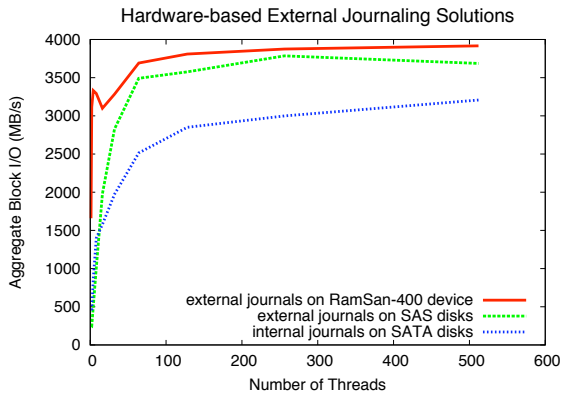


Figure 6. SAS disk, solid state disk external Lustre journaling and SATA disk internal journaling performances.

worst for almost all cases. External journals on a tier of SAS disks showed a gradual performance decrease for more than 256 I/O threads. External journals on the RamSan-400 device gave the best performance for all cases and this solution provided sustained performance with an increasing number of I/O threads. Overall, RamSan-based external journals achieved 3,292.6 MB/sec or 58.7% of our raw baseline performance. The performance dip for the RamSan-400 device at 16 threads was unexpected and is believed to be caused by queue starvation as a result of memory fragmentation pushing the SCSI commands beyond the scatter-gather limit. Unfortunately, we were unable to fully investigate this data point prior to losing access to the test platform and it should be noted that the 16 threads data point is outside of our normal operational envelope.

5 The Software Solution

As explained in Section3.2, Lustre's use of journals guarantees that when a client receives an RPC reply for a write request, the data is on stable storage and would survive a server crash. Although this implementation ensures data reliability, it serializes concurrent client write requests, as the currently running transaction cannot be closed and committed until the prior transaction fully commits to disk. With multiple RPCs in flight from the same client the overall operation flow would appear as if several concurrent write I/O RPCs arrive at the OST at the same time. In this case the serialization in the algorithm still exists, but with more requests coming in from different sources, the OST pipeline is more efficiently utilized. The OST will start its processing and then all these requests will block on waiting for their commits. Then, after each commit, replies for respective completed operations will be sent to the requesting client and then the client will send its next chunk of I/O requests. This algorithm works reasonably well from the aggregate bandwidth point of view as long as there are multiple writers that can keep the data flowing at all times. If there is only one client requesting service from a particular OST the inherent serialization in this algorithm is more pronounced; waiting for each transaction to commit introduces significant delay.

An obvious solution to this problem would be to send replies to clients immediately after the file data portion of a RPC is committed to disk. We have named this algorithm "*asynchronous journal commits*" and have implemented and tested this on our configuration.

Lustre's existing mechanism for metadata transactions allows it to send replies to clients about operation completion without waiting for data to be safe on disk. Every RPC reply from a server has a special field indicating the "id of the last transaction on stable storage" from that particular server's point of view. The client uses this information to keep a list of completed, but not committed operations, so that in case of a server crash these operations could be resent (replayed) to the server once the server resumes operations.

Our implementation extended this concept to write I/O RPCs on OSTs. In our implementation, dirty and flushed data pages are pinned in the client memory once they are submitted to the network. The client releases these pages only after it receives a confirmation from the OST indicating that the data was committed to stable storage.

In order to avoid using up all client memory with pinned data pages waiting for a confirmation for extended periods of time, upon receiving a reply with an uncommitted transaction id, a special "ping" RPC is scheduled on the client 7 seconds into the future (*ext3* flushes transactions to disk every 5 seconds). This "ping" RPC is pushed further in time if there are other RPCs scheduled by the client. This approach limits the impact to the client's memory footprint by bounding the time that uncommitted pages can remain outstanding. While the "ping" RPC is similar in nature to NFSv3's *commit* operation, Lustre optimizes this away in many cases by piggy-backing commit information on other RPCs destined for the same client-server pair.

The "*asynchronous journal commits*" algorithm results in a new set of steps taken by an OST processing a file update in the ordered journaling mode as detailed below. The following sequence of events is triggered by a Lustre client sending a write I/O request to an OST.

1. Server gets the destination object id and offset for this write operation.
2. Server allocates the necessary number of pages in memory and fetched the data from remote client into the pages via an RMA GET operation.
3. Server opens a transaction on the back-end file system.
4. Server updates file metadata, allocates blocks and extends the file size.
5. Server closes the transaction handle (not the JBD transaction) and if the RPC does NOT have the "async" flag set, then it obtains the wait handle.
6. Server writes pages with file data to disk synchronously.
7. If the "async" flag is set in the RPC, then Server completes the operation asynchronously.

   7a Server sends a reply to client.

   7b JBD then flushes the updated metadata blocks to the journaling device and writes the commit record.

8. If the "async" flag is NOT set in the RPC, then Server completes the operation synchronously.

   8a JBD flushes transaction closed in Step 5.

   8b Server sends a reply to the client that the operation was completed successfully.

The *obdfilter* benchmark was used for testing the asynchronous journal commit performance. Fig. 7 presents our results. The *ldiskfs* journal devices were created internally as part of each OST's block device. A single DDN S2A9900 couplet was used for this test. This approach resulted in dramatically fewer 4 KB updates (and associated head seeks) which substantially improved the aggregate performance to over 5,222.95 MB/s or 93% of our baseline performance. The dip at 16 threads is believed to be caused by the same mechanism as explained in the previous section and is outside of normal operational window.

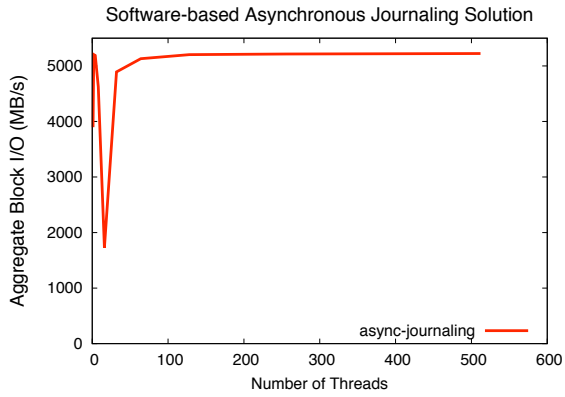


Figure 7. Asynchronous journaling performance

6 Results and Discussion

A comparative analysis of the hardware-based and software-based journaling methods is presented in Fig. 8. Please note that, the data presented in this figure is based on the data provided in figures 6 and 7. As can be seen, the software-based asynchronous journaling method clearly

outperforms the hardware-based solutions, providing virtually full baseline performance from the DDN S2A9900 couplet. One potential reason for the software-based solution outperforming the RamSan-based external journals may be the elimination of a network round-trip latency for each journal update operation as the journal resides on an SRP target separate from that of the block device in this configuration. Also, the performance of external journals on solid-state disks suggests that there may be other performance issues in the external journal code path which is encouraged by the lack of a performance improvement when asynchronous commits are used in combination with the RamSan-based external journal. The performance dip at 16 threads, present in both external journal and asynchronous journal methods, requires additional analysis.

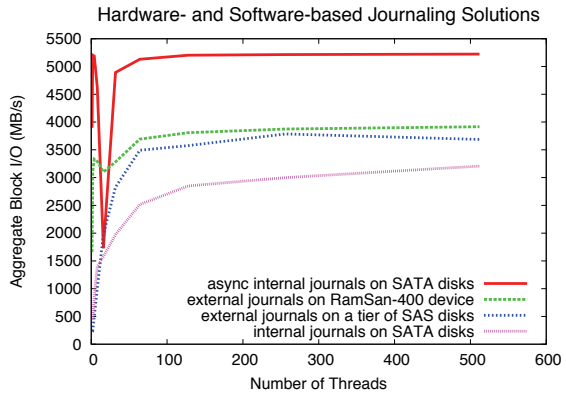


Figure 8. Aggregate Lustre performance with hardware- and software-based journaling methods.

The software-based asynchronous journaling method provides the best performance of the presented solutions, and does so at minimal cost. Therefore, we deployed this solution on Spider. We then analyzed the performance of Spider with the asynchronous journaling method on a real scientific application. For this purpose we used the Gyrokinetic Toroidal Code (GTC) application [15]. GTC is the most I/O intensive code running at scale (at the time of writing, the largest scale runs were at 120,000 cores on Jaguar) and is a 3D gyrokinetic Particle-In-Cell (PIC) code with toroidal geometry. It was developed at the Princeton Plasma Physics Laboratory (PPPL) and was designed to study turbulent transport of particles and energy in burning plasma. GTC is part of the US Department of Energy's Scientific Discovery through Advanced Computing (SciDAC) program. GTC is coded in standard Fortran 90/95 and MPI.

We used a version of GTC that has been modified to use the Adaptable IO System (ADIOS) I/O middleware [16] rather than standard Fortran I/O directives. ADIOS is developed by Georgia Tech and ORNL to manage I/O with a simple API and a supplemental, external configuration file. ADIOS has been implemented in several scientific production codes, including GTC. Earlier GTC tests with ADIOS on Jaguar XT4 showed increased scalability and higher performance when compared to the GTC runs with Fortran I/O. On the Jaguar XT4 segment, GTC with ADIOS achieved 75% of the maximum I/O performance measured with IOR [12].

Fig. 9 shows the GTC run times for 64 and 1,344 cores on Jaguar with and without asynchronous journals on Lustre file system. Both runs were configured with the same problem, and the difference in runtime can be attributed to the compute load of each core. During these runs, the observed I/O bandwidth by the application was increased by 56.3% on average and 64.8% when considering only the median values.

Translating the I/O bandwidth improvements to shorter runtimes will depend heavily on the I/O profile of the application and domain problem being investigated. In the 64 core case for GTC, the cores have a much larger compute load, and the percentage of runtime spent performing I/O drops from 6% to 2.6% when turning asynchronous journals on, with a 3.3% reduction in overall runtime. The 1,344 core test has much lighter compute load, and the runtime is dominated by I/O time – 70% of the runtime is I/O with synchronous journals, and 36% with asynchronous journals. This is reflected in the 49.5% reduction in overall runtime.

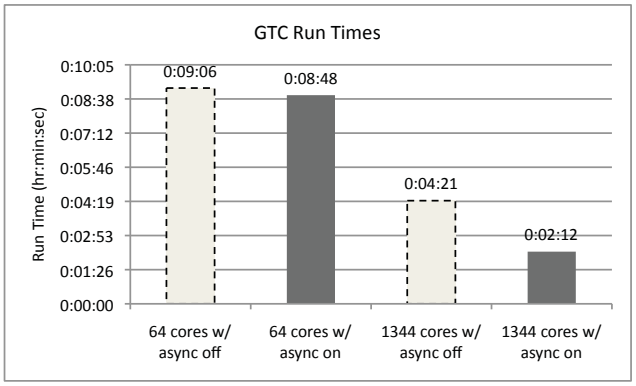


Figure 9. GTC run times for 64 and 1,344 cores on Jaguar with and without asynchronous journals.

Fig. 10 shows the histogram of I/O requests observed by the DDN S2A9900 during our GTC runs as a percent of total I/O requests observed. In this figure, "*Async Journals*" represents I/O requests observed when the asynchronous

journals were turned on and "*Sync Journals*" represents when asynchronous journals were turned off. Omitted request sizes from the graph account for less than 2.3% of the total I/O requests for the asynchronous journaling method and 0.76% for the synchronous journaling method. Asynchronous journaling clearly decreased the number of small I/O requests (0 to 127 KB) from 64% to 26.5%. This reduction minimized the disk head seeks, removed the serialization, and increased the overall disk I/O performance. Fig. 11 shows the same I/O request size histogram for 0 to 127 KB sized I/O requests as a percent of total I/O requests observed. Also in this figure "*Async Journals*" represents I/O requests observed when the asynchronous journals were turned on and "Sync Journals" represents when asynchronous journals were turned off. It can be seen that the asynchronous journaling method reduces the number of small I/O requests (0 to 128 KB) sent to the DDN controller (by delaying and aggregating the small journal commit requests into relatively larger but still small I/O requests, as explained in the previous section).

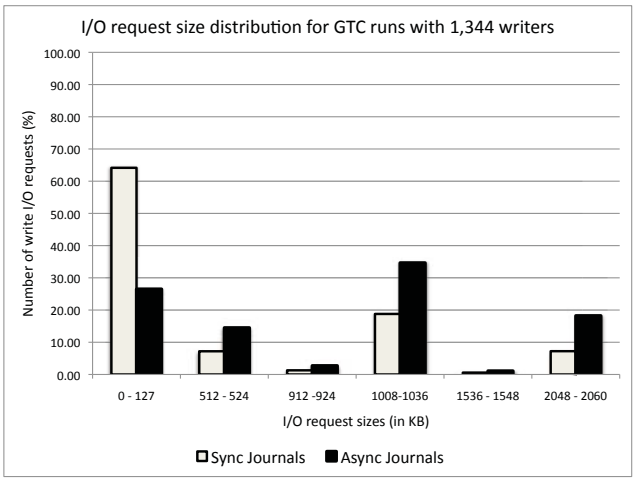


Figure 10. I/O request size histogram observed by the DDN S2A9900 controllers during the GTC runs.

Overall, our findings were motivated by the relatively modest IOPS performance (when compared to the bandwidth performance) of our DDN S2A9900 hardware. The DDN S2A9900 architecture uses "synchronous heads," or a variant of RAID3 that provides dual-failure redundancy. For a given LUN with 10 disks, a seek on the LUN requires a seek by all devices in the LUN. This approach provides highly optimized large I/O bandwidth, but it is not very efficient for small I/O. More traditional RAID5 and RAID6 implementations may not see the same speedup as the DDN hardware with our approach, as the stripes containing active journal data will likely remain resident in the controller

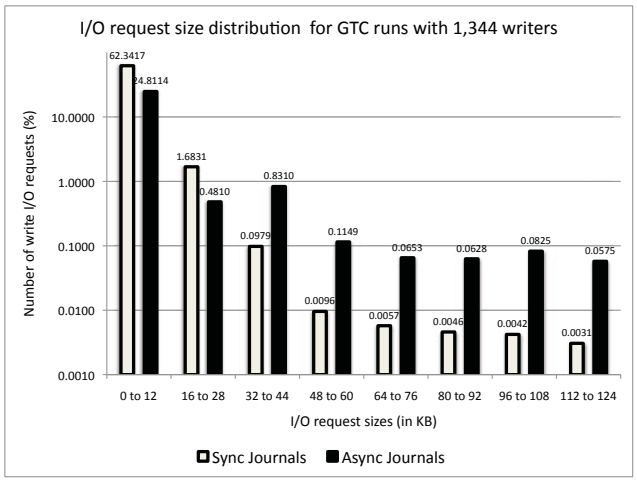


Figure 11. I/O request size histogram for 0 to 127 KB requests observed by the DDN S2A9900 controllers during the GTC runs.

cache, minimizing the need to do "read-modify-write" cycles to commit the journal records. Still, there will be head movement for those writes, which will incur a seek-penalty for the drive the stripe chunk that holds that portion of the journal. This will have an affect on the aggregate bandwidth of the RAID array. Some preliminary testing conducted by Sun Microsystems using their own RAID hardware has shown improved performance, but the details of that testing is not currently public. We did not have the chance to test our approach on non-DDN hardware, and are unable to further qualify the impact of our solution on other RAID controllers at this time.

Our approach removed the bottleneck out of the critical write path by providing an asynchronous write/commit mechanism for the Lustre file system. This solution has been previously proposed by NFSv3 and others, and we were able to implement it in an efficient manner to boost our write performance in a very large scale production storage deployment. Our approach comes with a temporary increase in memory consumption on clients, which we believe is a fair price for the performance increases. Our changes are restricted to how Lustre uses the journal, and not the operation of the journal itself. Specifically, we do not wait for the journal commit prior to allowing the client to send more data. As we have not told the client that the data is stable, it will retain it in the event the OSS (OST) dies and the client needs to replay its I/O requests. The guarantees about file system consistency at the local OST remain unchanged. Also, our limited tests with manually injected power failures on the server side with active write/modify I/O client RPCs in flight provided consistent data on the file system, provided the clients successfully completed recovery.

7 Conclusions

Initial IOR testing with Spider's DDN S2A9900s and SATA drives on Jaguar showed that Lustre level write performance was 24.9% of the baseline performance with a 1 MB transfer size. Profiling the I/O stream using the DDN utilities revealed a large number of 4 KB writes in addition to the expected 1 MB writes. These small writes were traced to *ldiskfs* journal updates. This information allowed us to identify bottlenecks in the way Lustre was using the journal – each batch of write requests blocked on the commit of a journal transaction, which added serialization to the request stream and incurred the latency of a disk head seek for each write.

We developed and implemented both a hardware based solution as well as a software solution to these issues. We used external journals on solid state devices to eliminate head seeks for the journal, which allowed us to achieve 3,292.6 MB/sec or 58.7% of our baseline performance per DDN S2A9900. By removing the requirement for a synchronous journal commit for each batch of writes, we observed dramatically fewer 4 KB journal updates (up to 37%) and associated head seeks. This substantially improved our block I/O performance to over 5,222.95 MB/s or 93% of our baseline performance per DDN S2A9900 couplet.

Tests with a real-world scientific application such as GTC have shown an average I/O bandwidth improvement of 56.3%. Overall, asynchronous journaling has proven to be a highly efficient solution to our performance problem in terms of performance as well as cost-effectiveness.

Our approach removed a bottleneck from the critical write path by providing an asynchronous write/commit mechanism for the Lustre file system. This solution has been previously proposed for NFSv3 and other file systems, and we were able to implement it in an efficient manner to significantly boost our write performance in a very large scale production storage deployment.

Our current understanding and testing show that our approach does not change the guarantees of file system consistency at the local OST level, as the modifications only affect how Lustre uses the journal, and not the operation of the journal itself. However, this approach comes with a temporary increase of memory consumption on clients while waiting for the server to commit the transactions. We find this a fair exchange for the substantial performance enhancement it provides on our very large scale production parallel file system.

Our approach and findings are likely not specific to our DDN hardware, and are of interest to developers and large-scale HPC vendors and integrators in our community. Future work will include verifying broad applicability as test hardware becomes available. Other potential future work includes an analysis of how other scalable parallel file systems, such as IBM's GPFS, approach the synchronous write performance penalties.

8 Acknowledgements

The authors would like to thank our colleagues at the National Center for Computational Sciences at Oak Ridge National Laboratory for their support of our work, with special thanks to Scott Klasky for his help with the GTC code and Youngjae Kim and Douglas Fuller for corrections and suggestions.

The research was sponsored by the Mathematical, Information, and Computational Sciences Division, Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract No. DE-AC05-00OR22725 with UT-Battelle, LLC.

References

[1] S. R. Alam, R. F. Barrett, M. R. Fahey, J. A. Kuehn, J. M. Larkin, R. Sankaran, and P. H. Worley. Cray XT4: An early evaluation for petascale scientific simulation. In *Proceedings of the ACM/IEEE conference on High Performance Networking and Computing (SC07)*, Reno, NV, 2007.

[2] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan. Scalable i/o forwarding framework for high-performance computing systems. In *Proceedings of the IEEE International Conference on Cluster Computing*, Aug, 2009.

[3] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer. Non-volatile memory for fast, reliable file systems. In *Proceedings of the 5th ASPLOS*, pages 10–22, 1992.

[4] R. Brightwell, K. Pedretti, and K. D. Underwood. Initial performance evaluation of the cray seastar interconnect. In *HOTI '05: Proceedings of the 13th Symposium on High Performance Interconnects*, pages 51–57, Washington, DC, USA, 2005. IEEE Computer Society.

[5] Cray Inc. Cray XT5. `http://cray.com/Products/XT/Systems/XT5.aspx`.

[6] Data Direct Networks. DDN S2A9900. `http://www.ddn.com/9900`.

[7] Dell. Dell PowerEdge 1950 Server. `http://www.dell.com/downloads/global/products/pedge/en/1950_specs.pdf`.

[8] J. Dongarra, H. Meuer, and E. Strohmaier. Top500 November 2009 List. `http://www.top500.org/lists/2009/11`, 2008.

[9] J. Dongarra, H. Meuer, and E. Strohmaier. Top500 supercomputing sites. `http://www.top500.org`, 2009.

[10] G. R. Ganger and Y. N. Patt. Metadata update performance in file systems. In *OSDI '94: Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, page 5, Berkeley, CA, USA, 1994. USENIX Association.

[11] ioperformance.com. xdd performance benchmark, version 6.5. `http://www.ioperformance.com`, 2008.

[12] S. Klasky. private communication, Sept. 2009.

[13] A. Leventhal. Hybrid storage pools in the 7410. `http://blogs.sun.com/ahl/entry/fishworks_launch`.

[14] A. Leventhal. Flash storage memory. *Communications of the ACM*, 51(7):47–51, 2008.

[15] Z. Lin, T. S. Hahm, W. W. Lee, W. M. Tang, , and R. B. White. Turbulent transport reduction by zonal flows: Massively parallel simulations. *Science*, 18:1835–1837, 1988.

[16] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan. Adaptable, metadata rich io methods for portable high performance io. In *In Proceedings of IPDPS'09, May 25-29, Rome, Italy*, 2009.

[17] R. Mendel and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, 1992.

[18] D. A. Nowak and M. Seagar. ASCI terascale simulation: Requirements and deployments. `http://www.ornl.gov/sci/optical/docs/Tutorial19991108Nowak.pdf`.

[19] Oak Ridge National Laboratory, National Center for Computational Sciences. Jaguar. `http://www.nccs.gov/jaguar/`.

[20] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS Version 3 - Design and Implementation. *Proceedings of the Summer 1994 USENIX Technical Conference*, pages 137–152, 1994.

[21] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis and evolution of journaling file systems. In *Proceedings of the Annual USENIX Technical Conference*, May 2005.

[22] B. Schroeder and G. A. Gibson. Understanding failures in petascale computers. *Journal of Physics Conference Series*, 78(1):012022–+, July 2007.

[23] M. Seltzer, G. Ganger, K. McKusick, K. Smith, C. Soules, and C. Stein. Journaling versus Soft Updates: Asynchronous Meta-data Protection in File Systems. In *Proceedings of the USENIX Technical Conference*, pages 71–84, June 2000.

[24] H. Shan and J. Shalf. Using IOR to analyze the I/O performance of XT3. In *Proceedings of the 49th Cray User Group (CUG) Conference 2007*, Seattle, WA, 2007.

[25] G. Shipman. Spider and SION: Supporting the I/O Demands of a Peta-scale Environment. In *Cray User Group Meeting*, 2008.

[26] G. Shipman, D. Dillow, S. Oral, and F. Wang. The spider center wide file system: From concept to reality. In *Proceedings,Cray User Group (CUG) Conference, Atlanta, GA*, May 2009.

[27] Sun Microsystems. Lustre i/o kit, obdfilter-survey. `http://manual.lustre.org/manual/LustreManual16_HTML/LustreIOKit.html`.

[28] Sun Microsystems Inc. Luste wiki. `http://wiki.lustre.org`, 2009.

[29] Texas Memory Systems Inc. Ramsan-400. `http://www.ramsan.com/products/ramsan-400.htm`.

[30] S. C. Tweedie. Journaling the Linux ext2fs Filesystem. In *Proceedings of the fourth annual Linux expo*, 1998.

[31] F. Wang, S. Oral, G. Shipman, O. Drokin, T. Wang, and I. Huang. Understanding lustre filesystem internals. Technical Report ORNL/TM-2009/117, Oak Ridge National Lab., National Center for Computational Sciences, 2009.

[32] W. Yu, S. Oral, S. Canon, J. Vetter, and R. Sankaran. Empirical analysis of a large-scale hierarchical storage system. In *14th European Conference on Parallel and Distributed Computing (Euro-Par 2008)*, 2008.

[33] W. Yu, J. Vetter, and S. Oral. Performance characterization and optimization of parallel I/O on the Cray XT. In *Proceedings of 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS'08)*, Miami, FL, 2008.

Panache: A Parallel File System Cache for Global File Access

Marc Eshel Roger Haskin Dean Hildebrand Manoj Naik Frank Schmuck
Renu Tewari
IBM Almaden Research
{eshel, roger, manoj, schmuck}@almaden.ibm.com, {dhildeb, tewarir}@us.ibm.com

Abstract

Cloud computing promises large-scale and seamless access to vast quantities of data across the globe. Applications will demand the reliability, consistency, and performance of a traditional cluster file system regardless of the physical distance between data centers.

Panache is a scalable, high-performance, clustered file system cache for parallel data-intensive applications that require wide area file access. Panache is the first file system cache to exploit parallelism in every aspect of its design—parallel applications can access and update the cache from multiple nodes while data and metadata is pulled into and pushed out of the cache in parallel. Data is cached and updated using pNFS, which performs parallel I/O between clients and servers, eliminating the single-server bottleneck of vanilla client-server file access protocols. Furthermore, Panache shields applications from fluctuating WAN latencies and outages and is easy to deploy as it relies on open standards for high-performance file serving and does not require any proprietary hardware or software to be installed at the remote cluster.

In this paper, we present the overall design and implementation of Panache and evaluate its key features with multiple workloads across local and wide area networks.

1 Introduction

Next generation data centers, global enterprises, and distributed cloud storage all require sharing of massive amounts of file data in a consistent, efficient, and reliable manner across a wide-area network. The two emerging trends of offloading data to a distributed storage cloud and using the MapReduce [11] framework for building highly parallel data-intensive applications, have highlighted the need for an extremely scalable infrastructure for moving, storing, and accessing massive amounts of data across geographically distributed sites. While large cluster file systems, e.g., GPFS [26], Lustre [3], PanFS [29] and Internet-scale file systems, e.g., GFS [14], HDFS [6] can scale in capacity and access bandwidth to support a large number of clients and petabytes of data, they cannot mask the latency and fluctuating performance of accessing data across a WAN.

Traditionally, NFS (for Unix) and CIFS (for Windows) have been the protocols of choice for remote file serving. Originally designed for local area access, both are rather "chatty" and therefore unsuited for wide-area access. NFSv4 has numerous optimizations for wide-area use, but its scalability continues to suffer from the "single server" design. NFSv4.1, which includes pNFS, improves I/O performance by enabling parallel data transfers between clients and servers. Unfortunately, while NFSv4 and pNFS can improve network and I/O performance, they cannot completely mask WAN latencies nor operate during intermittent network outages.

As "storage cloud" architectures evolve from a single high bandwidth data-center towards a larger multi-tiered storage delivery architecture, e.g., Nirvanix SDN [7], file data needs to be efficiently moved across locations and be accessible using standard file system APIs. Moreover, for data-intensive applications to function seamlessly in "compute clouds", the data needs to be cached closer to or at the site of the computation. Consider a typical multi-site compute cloud architecture that presents a virtualized environment to customer applications running at multiple sites within the cloud. Applications run inside a virtual machine (VM) and access data from a virtual LUN, which is typically stored as a file, e.g., VMware's .vmdk file, in one of the data centers. Today, whenever a new virtual machine is configured, migrated, or restarted on failure, the OS image and its virtual LUN (greater than 80 GB of data) must be transferred between sites causing long delays before the application is ready to be online. A better solution would store all files at a central core site and then dynamically cache the OS image and its virtual LUN at an edge site closer to the physical machine. The machine hosting the VMs (e.g., the ESX server) would connect to the edge site to access the virtual LUNs over NFS while the data would move transparently between the core and edge sites on demand. This enormously simplifies both the time and complexity of configuring new VMs and dynamically moving them across a WAN.

Research efforts on caching file system data have mostly been limited to improving the performance of a single client machine [18, 25, 22]. Moreover, most available solutions are NFS client based caches [15, 18]

and cannot function as a standalone file system (without network connectivity) that can be used by a POSIX-dependent application. What is needed is the ability to pull and push data in parallel, across a wide-area network, store it in a scalable underlying infrastructure while guaranteeing file system consistency semantics.

In this paper we describe Panache, a read-write, multi-node file system cache built for scalability and performance. The distributed and parallel nature of the system completely changes the design space and requires re-architecting the entire stack to eliminate bottlenecks. The key contribution of Panache is a *fully parallelizable* design that allows every aspect of the file system cache to operate in parallel. These include:

- *parallel ingest* wherein, on a miss, multiple files and multiple chunks of a file are pulled into the cache in parallel from multiple nodes,
- *parallel access* wherein a cached file is accessible immediately from all the nodes of the cache,
- *parallel update* where all nodes of the cache can write and queue, for remote execution, updates to the same file in parallel or update the data and metadata of multiple files in parallel,
- *parallel delayed data write-back* wherein the written file data is asynchronously flushed in parallel from multiple nodes of the cache to the remote cluster, and
- *parallel delayed metadata write-back* where all metadata updates (file creates, removes etc.) can be made from any node of the cache and asynchronously flushed back in parallel from multiple nodes of the cache. The multi-node flush preserves the order in which dependent operations occurred to maintain correctness.

There is, by design, no single metadata server and no single network end point to limit scalability as is the case in typical NAS systems. In addition, all *data and metadata updates* made to the cache are *asynchronous*. This is essential to support WAN latencies and outages as high performance applications cannot function if every update operation requires a WAN round-trip (with latencies running from 30ms to more than 200ms).

While the focus in this paper is on the parallel aspects of the design, Panache is a fully functioning POSIX-compliant caching file system with additional features including disconnected operations, persistence across failures, and consistency management, that are all needed for a commercial deployment. Panache also borrows from Coda [25] the basic premise of conflict handling and conflict resolution when supporting disconnected mode operations and manages them in a clustered setting. However, these are beyond the scope of this paper. In this paper, we present the overall design and implementation of Panache and evaluate its key features with multiple workloads across local and wide area networks.

The rest of the paper is organized as follows. In the next two sections we provide a brief background of pNFS and GPFS, the two essential components of Panache. Section 4 provides an overview of the Panache architecture. The details of how synchronous and asynchronous operations are handled are described in Section 5 and Section 6. Section 7 presents the evaluation of Panache using different workloads. Finally, Section 8 discusses the related work and Section 9 presents our conclusions.

2 Background

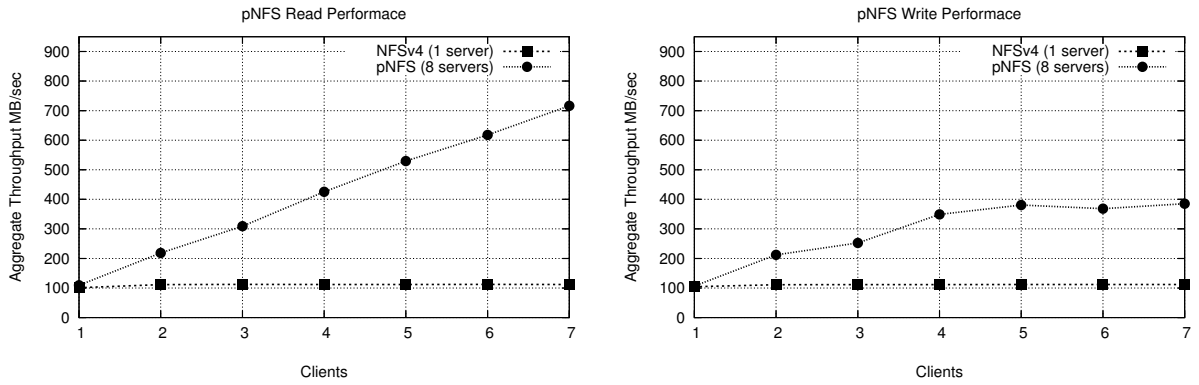
In order to better understand the design of Panache let us review its two basic components: GPFS, the parallel cluster file system used to store the cached data, and pNFS, the nascent industry-standard protocol for transferring data between the cache and the remote site.

GPFS: General Parallel File System [26] is IBM's high-performance shared-disk cluster file system. GPFS achieves its extreme scalability through a shared-disk architecture. Files are wide-striped across all disks in the file system where the number of disks can range from tens to several thousand disks in the largest GPFS installations. In addition to balancing the load on the disks, striping achieves the full throughput that the disk subsystem is capable of by reading and writing data blocks in parallel.

The switching fabric that connects file system nodes to disks may consist of a storage area network (SAN), e.g., Fibre Channel, iSCSI, or, a general-purpose network by using I/O server nodes. GPFS uses distributed locking to synchronize access to shared disks where all nodes share responsibility for data and metadata consistency. GPFS distributed locking protocols ensure file system consistency is maintained regardless of the number of nodes simultaneously reading from and writing to the file system, while at the same time allowing the parallelism necessary to achieve maximum throughput.

pNFS: The pNFS protocol, now an integral part of NFSv4.1, enables clients for direct and parallel access to storage while preserving operating system, hardware platform, and file system independence [16]. pNFS clients and servers are responsible for control and file management operations, but delegate I/O functionality to a storage-specific layout driver on the client.

To perform direct and parallel I/O, a pNFS client first requests layout information from a pNFS server. A layout contains the information required to access any byte of a file. The layout driver uses the information to translate I/O requests from the pNFS client into I/O requests



(a) pNFS Reads (b) pNFS Writes

Figure 1: **pNFS Read and Write performance.** *pNFS performance scales with available hardware and network bandwidth while NFSv4 performance remains constant due to the single server bottleneck.*

directed to the data servers. For example, the NFSv4.1 file-based storage protocol stripes files across NFSv4.1 data servers, with only READ, WRITE, COMMIT, and session operations sent on the data path. The pNFS metadata server can generate layout information itself or request assistance from the underlying file system.

3 pNFS for Scalable Data Transfers

Panache leverages pNFS to increase the scalability and performance of data transfers between the cache and remote site. This section describes how pNFS performs in comparison to vanilla NFSv4.

NFS and CIFS have become the de-facto file serving protocols and follow the traditional multiple client–single server model. With the single-server design, which binds one network endpoint to all files in a file system, the back-end cluster file system is exported by a single NFS server or multiple independent NFS servers.

In contrast, pNFS removes the single server bottleneck by using the storage protocol of the underlying cluster file system to distribute I/O across the bisectional bandwidth of the storage network between clients and data servers. In combination, the elimination of the single server bottleneck and direct storage access by clients yields superior remote file access performance and scalability [16].

Figure 2 displays the pNFS-GPFS architecture. The nodes in the cluster exporting data for pNFS access are divided into (possibly overlapping) groups of state and data servers. pNFS client metadata requests are partitioned among the available state servers while I/O is distributed across all of the data servers. The pNFS client requests the data layout from the state server using a LAYOUTGET operation. It then accesses data in parallel by using the layout information to send NFSv4 READ and WRITE operations to the correct data servers. For writes, once the I/O is complete, the client

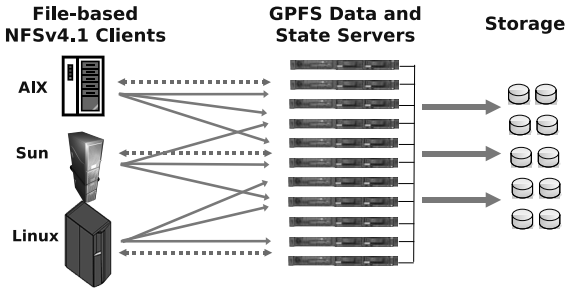
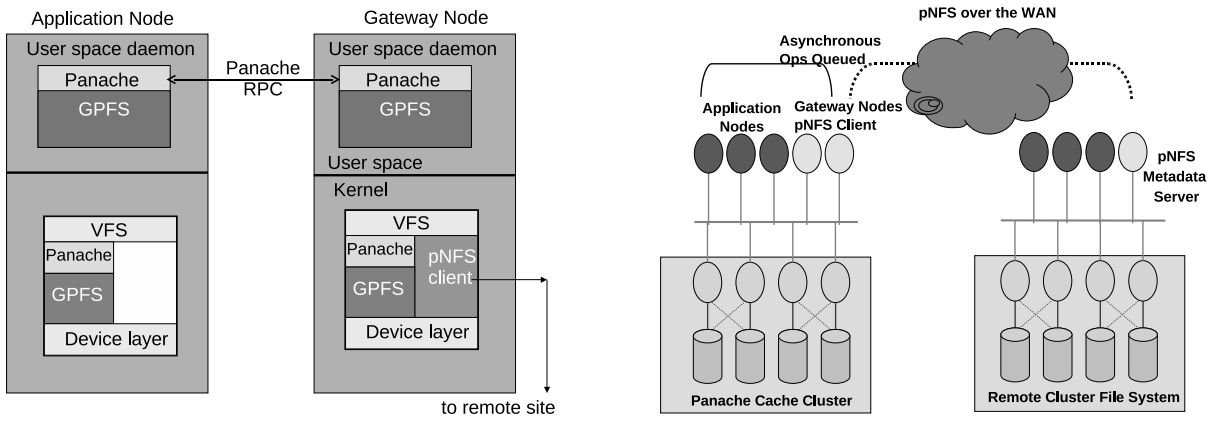


Figure 2: **pNFS-GPFS Architecture.** *Servers are divided into (possibly overlapping) groups of state and data servers. pNFS/NFSv4.1 clients use the state servers for metadata operations and use the file-based layout to perform parallel I/O to the data servers.*

sends an NFSv4 COMMIT operation to the state server. This single COMMIT operation flushes data to stable storage on every data server. The underlying cluster file system management protocol maintains the freshness of NFSv4 state information among servers.

To demonstrate the effectiveness of pNFS for scalable file access, Figures 1(a) and 1(b) compare the aggregate I/O performance of pNFS and standard NFSv4 exporting a seven server GPFS file system. GPFS returns a file layout to the pNFS client that stripes files across all data servers using a round-robin order and continually alternates the first data server of the stripe. Experiments use the IOR micro-benchmark [2] to increase the number of clients accessing individual large files. As the number of NFSv4 clients accessing a single NFSv4 server is increased, performance remains constant. On the other hand, pNFS can better utilize the available bandwidth. With reads, pNFS clients completely saturate the local network bandwidth. Write throughput ascends to 3.8x of standard NFSv4 performance with five clients before reaching the limitations of the storage controller.



(a) Node Block Diagram

(b) Cache Cluster Architecture

Figure 3: **Panache Caching Architecture.** *(a) Block diagram of an application and gateway node. On tje gateway node, Panache communicates with the pNFS client kernel module through the VFS layer. The application and gateway nodes communicate via custom RPCs through the user-space daemon. (b) The cache cluster architecture. The gateway nodes of the cache cluster act as pNFS/NFS clients to access the data from the remote cluster. The application nodes access data from the cache cluster.*

4 Panache Architecture Overview

The design of the Panache architecture is guided by the following performance and operational requirements:

- Data and metadata read performance, on a cache hit, matches that of a cluster file system. Thus, reads should be limited only by the aggregate disk bandwidth of the local cache site and not by the WAN.
- Read performance, on a cache miss, is limited only by the network bandwidth between the sites.
- Data and metadata update performance matches that of a cluster file system update.
- The cache can operate as a standalone fileserver (in the presence of intermittent or no network connectivity), ensuring that applications continue to see a POSIX compliant file system.

Panache is implemented as a multi-node caching layer, integrated within the GPFS, that can persistently and consistently cache *data and metadata* from a remote cluster. Every node in the Panache cache cluster has direct access to cached data and metadata. Thus, once data is cached, applications running on the Panache cluster achieve the same performance as if they were running directly on the remote cluster. If the data is not in the cache, Panache acts as a caching proxy to fetch the data in parallel both by using a parallel read across multiple cache cluster nodes to drive the ingest, and from multiple remote cluster nodes using pNFS. Panache allows updates to be made to the cache cluster at local cluster performance by asynchronously pushing all updates of data and metadata to the remote cluster.

More importantly, Panache, compared to other single-node file caching solutions, can function both as a standalone clustered file system and as a clustered caching proxy. Thus applications can run on the cache cluster using POSIX semantics and access, update, and traverse the directory tree even when the remote cluster is offline. As the cache mimics the same namespace as the remote cluster, browsing through the cache cluster (say with ls -R) shows the same listing of directories and files, as well as most of their remote attributes. Furthermore, NFS/pNFS clients can access the cache and see the same view of the data (as defined by NFS consistency semantics) as NFS clients accessing the data directly from the remote cluster. In essence, both in terms of consistency and performance, applications can operate as if the WAN did not exist.

Figure 3(b) shows the schematic of the Panache architecture with the cache cluster and the remote cluster. The remote cluster can be any file system or NAS filer exporting data over NFS/pNFS. Panache can operate on a multi-node cluster (henceforth called the cache cluster) where all nodes need not be identical in terms of hardware, OS, or support for remote network connectivity. Only a set of designated nodes, called *Gateway nodes*, need to have the hardware and software support for remote access. These nodes internally act as NFS/pNFS client proxies to fetch the data in parallel from the remote cluster. The remaining nodes of the cluster, called *Application nodes*, service the application data requests from the Panache cluster. The split between application and gateway nodes is conceptual and any node in the cache cluster can function both as a gateway node or a application node based on its configuration. The gate-

way nodes can be viewed as the edge of the cache cluster that can communicate with the remote cluster while the application nodes interface with the application. Figure 3(a) illustrates the internal components of a Panache node. Gateway nodes communicate with the pNFS kernel module via the VFS layer, which in turn communicates with the remote cluster. Gateway and application nodes communicate with each other via 26 different internal RPC requests from the user space daemon.

When an application request cannot be satisfied by the cache, due to a cache miss or to invalid cached data, the application node sends a read request to one of the gateway nodes. The gateway node then accesses the data from the remote cluster and returns it to the application node. Panache supports different mechanisms for gateway nodes to share the data with application nodes. One option is for the gateway nodes to write the remote data to the shared storage, which the application nodes can then read and return the data to the application. Another option is for gateway nodes to transfer the data directly to the application nodes using the cluster interconnect. Our current Panache prototype shares data through the storage subsystem, which can generally give higher performance than a typical network link.

All updates to the cache cause an application node to send and queue a command message on one or more gateway nodes. Note that this message includes no file data or metadata. At a later time, the gateway node(s) will read the data in parallel from the storage system and push it to the remote cluster over pNFS.

The selection of a gateway node to service a request needs to ensure that dependent requests are executed in the intended order. The application node selects a gateway node using a hash function based on a unique identifier of the object on which a file system operation is requested. Sections 5 and 6 describe how this identifier is chosen and how Panache executes read and update operations in more detail.

4.1 Consistency

Consistency in Panache can be controlled across various dimensions and can be defined relative to the cache cluster, the remote cluster and the network connectivity.

Definition 1 Locally consistent: *The cached data is considered locally consistent if a read from a node of the cache cluster returns the last write from any node of the cache cluster.*

Definition 2 Validity Lag: *The time delay between a read at the cache cluster reflecting the last write at the remote cluster.*

Definition 3 Synchronization Lag: *The time delay between a read at the remote cluster reflecting the last write at the cache cluster.*

Definition 4 Eventually Consistent: *After recovering from a node or network failure, in the absence of further failures, the cache and remote cluster data will eventually become consistent within the bounds of the lags.*

Panache, by virtue of relying on the cluster-wide distributed locking mechanism of the underlying clustered file system, is always locally consistent for the updates made at the cache cluster. Accesses are serialized by electing one of the nodes to be the token manager and issuing read and write tokens [26]. Local consistency within the cache cluster basically translates to the traditional definition of strong consistency [17].

For cross-cluster consistency across the WAN, Panache allows both the validity lag and the synchronization (synch) lag to be tunable based on the workload. For example, setting the validity lag to zero ensures that data is always validated with the remote cluster on an open and setting the synch lag to zero ensures that updates are flushed to the remote cluster immediately.

NFS uses a attribute timeout value (typically 30s) to recheck with the server if the file attributes have changed. Dependence on NFS consistency semantics can be removed via the O_DIRECT parameter (which disables NFS client data caching) and/or by disabling attribute caching (effectively setting the attribute timeout value to 0). NFSv4 file delegations can reduce the overhead of consistency management by having the remote cluster's NFS/pNFS server transfer ownership of a file to the cache cluster. This allows the cache cluster to avoid periodically checking the remote file's attributes and safely assume that the data is valid.

When the synch lag is greater than zero, all updates made to the cache are asynchronously committed at the remote cluster. In fact, the semantics will no longer be close-to-open as updates will ignore the file close and will be time delayed. Asynchronous updates can result in conflicts which, in Panache, are resolved using policies as discussed in Section 6.3.

When there is a network or remote cluster failure both the validation lag and synch lag become indeterminate. When connectivity is restored, the cache and remote clusters are eventually synchronized.

5 Synchronous Operations

Synchronous operations block until the remote operation completes, either because an object does not exist in the cache, i.e., a cache miss, or the object exists in the cache but needs to be revalidated. In either case, the object or its attributes need to be fetched or validated from the remote cluster on an application request. All file system data and metadata "read" operations, e.g., *lookup, open, read, readdir, getattr*, are synchronous. Unlike typical caching systems, Panache ingests the data and metadata

in parallel from multiple gateway nodes so that the cache miss or pre-populate time is limited only by the network bandwidth between the caching and remote clusters.

5.1 Metadata Reads

The first time an application node accesses an object via the VFS *lookup* or *open* operations, the object is created in the cache cluster as an empty object with no data. The mapping with the remote object is through the NFS filehandle that is stored with the inode as an extended attribute. The flow of messages proceeds as follows: i) the application node sends a request to the designated gateway node based on a hash of the inode number or its parent inode number if the object doesn't exist ii) the gateway node sends a request to the remote cluster's NFS/pNFS server(s), iii) on success at the remote cluster, the filehandle and attributes of the object are returned back to the gateway node which then creates the object in the cache, marks it as empty, and stores the remote filehandle mapping, iv) the gateway node then returns success back to the application node. On a later read or prefetch request the data in the empty object will be populated.

5.2 Parallel Data Reads

On an application *read* request, the application node first checks if the object exists in the local cache cluster. If the object exists but is empty or incomplete, the application node, as before, requests the designated gateway node to read in the requested offset and size. The gateway node, based on the prefetch policy, fetches the requested bytes or the entire file and writes it to the cache cluster. With prefetching, the whole file is asynchronously read after the byte-range requested by the application is ingested. Panache supports both whole file and partial file (segments consisting of a set of contiguous blocks) caching. Once the data is ingested, the application node reads the requested bytes from the local cache and returns them to the application as if they were present locally all along. Recall that the application and gateway nodes exchange only request and response messages while the actual data is accessed locally via the shared storage subsystem. On a later cache hit, the application node(s) can directly service the file read request from the local cache cluster. The cache miss performance is, therefore, limited by the network bandwidth to the remote cluster, while the cache hit performance is limited only by the local storage subsystem bandwidth (as shown in Table 1).

Panache scales I/O performance by using multiple gateway nodes to read chunks of a single file in parallel from the multiple nodes over NFS/pNFS. One of the gateway nodes (based on the hash function) becomes the coordinator for a file. It, in turn, divides the requests among the other gateway nodes which can proceed to read the data in parallel. Once a node is finished with its chunk it requests the coordinator for more chunks to read. When all the requested chunks have been read the gateway node responds to the application node that the requested blocks of the object are now in cache. If the remote cluster file system does not support pNFS but does support NFS access to multiple servers, data can still be read in parallel. Given N gateway nodes at the cache cluster and M nodes exporting data at the remote cluster, a file can be read either in 1xM (pNFS case) parallel streams, or min{N,M} 1x1 parallel streams (multiple gateway parallel reads with NFS) or NxM parallel streams (multiple gateway parallel reads with pNFS) as shown in Figure 4.

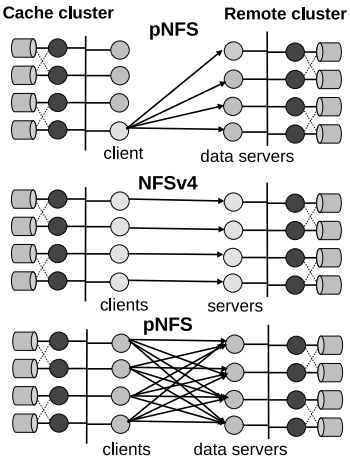


Figure 4: **Multiple gateway node configurations.** *The top setup is a single pNFS client reading a file from multiple data servers in parallel. The middle setup is multiple gateway nodes acting as NFS clients reading parts of the file from the remote cluster's NFS servers. The bottom setup has multiple gateway nodes acting as pNFS clients reading parts of the file in parallel from multiple data servers.*

File Read	2 gateway nodes	3 gateway nodes
Miss	1.456 Gb/s	1.952 Gb/s
Hit	8.24 Gb/s	8.24 Gb/s
Direct over pNFS	1.776 Gb/s	2.552 Gb/s

Table 1: **Panache (with pNFS) and pNFS read performance using the IOR benchmark.** *Clients read 20 files of 5GB each using 2 and 3 gateway nodes with gigabit ethernet connecting to a 6-node remote cluster. Panache scales on both cache miss and cache hit. On cache miss, Panache incurs the overhead of passing data through the SAN, while on a cache hit it saturates the SAN.*

5.3 Namespace Caching

Panache provides a standard POSIX file system interface for applications. When an application tra-

verses the namespace directory tree, Panache reflects the view of the corresponding tree at the remote cluster. For example, an "ls -R" done at the cache cluster presents the same list of entries as one done at the remote cluster. Note that Panache does not simply return the directory listing with $dirents$ containing the $< name, inode\_num >$ pairs from the remote cluster (as an NFS client would). Instead, Panache first creates the directory entries in the local cluster and then returns the cached name and inode number to the application. This is done to ensure application nodes can continue to traverse the directory tree if a network or server outage occurs. In addition, if the cache simply returns the remote inode numbers to the application, and later a file is created in the cache with that inode number, the application may observe different inode numbers for the same file.

One approach to returning consistent inode numbers to the application on a *readdir* (directory listing) or *lookup* and *getattr*, e.g., file stat, is by mandating that the remote cluster and the cache cluster mirror the same inode space. This can be impossible to implement where remote inode numbers conflict with inode numbers of reserved files and clearly limits the choice of the remote cluster file systems. A simple approach is to fetch the attributes of all the directory entries, i.e., an extra lookup across the network and create the files locally on a *readdir* request. This approach of creating files on a directory access has an obvious performance penalty for directories with a large number of files.

To solve the performance problems with creates on a *readdir* and allow for the cache cluster to operate with a separate inode space, we create only the directory entries in the local cluster and create placeholders for the actual files and directories. This is done by allocating but not creating or using inodes for the new entries. This allows us to satisfy the readdir request with locally allocated inode numbers without incurring the overhead of creating all the entries. These allocated, but not yet created, entries are termed *orphans*. On a subsequent *lookup*, the allocated inode is "filled" with the correct attributes and created on disk. Orphan inodes cause interesting problems on *fsck*, file deletes, and cache eviction and have to be handled separately in each case. Table 2 shows the performance (in secs) of reading a directory for 3 cases: i) where the files are created on a *readdir*, ii) when only orphan inodes are created, and iii) when the readdir is returned locally from the cache.

5.4 Data and Attribute Revalidation

The data validity in the cache cluster is controlled by a revalidation timeout, in a manner similar to the NFS attribute timeout, whose value is determined by the desired *validity lag* of the workload. The cache cluster's inode stores both the local modification time $mtime_{local}$ and inode change time $ctime_{local}$ along with the remote $mtime_{remote}, ctime_{remote}$. When the object is accessed after the revalidation timeout has expired the gateway node gets the remote object's time attributes and compares them with the stored values. A change in $mtime_{remote}$ indicates that the object's data was modified and a change in $ctime_{remote}$, indicates that the object's inode was changed as the attributes or data was modified [1]. In case the remote cluster supports NFSv4 with delegations, some of this overhead can be removed by assuming the data is valid when there is an active delegation. However, every time the delegation is recalled, the cache falls back to timeout based revalidation.

Files per dir	readdir & creates	readdir & orphan inodes	readdir from cache
100	1.952 (s)	0.77 (s)	0.032 (s)
1,000	3.122	1.26	0.097
10,000	7.588	2.825	0.15
100,000	451.76	25.45	1.212

Table 2: **Cache traversal with a readdir.** *Performance (in secs.) of a readdir on a cache miss where the individual files are created vs. the orphan inodes. The last column shows the performance of readdir on a cache hit.*

During a network outage or remote server failure, the revalidation lag becomes indeterminate. By policy, either the requests are made blocking where they wait till connectivity is restored or all synchronous operations are handled locally by the cache cluster and no request is sent to the gateway node for remote execution.

6 Asynchronous Operations

One important design decision in Panache was to mask the WAN latencies by ensuring applications see the cache cluster's performance on all data writes and metadata updates. Towards that end, all data writes and metadata updates are done asynchronously—the application proceeds after the update is "committed" to the cache cluster with the update being pushed to the remote cluster at a later time governed by the synch lag. Moreover, executing updates to the remote cluster is done in parallel across *multiple* gateway nodes. Most caching systems delay only data writes and perform all the metadata and namespace updates synchronously, preventing disconnected operation. By allowing asynchronous metadata updates, Panache allows data and metadata updates at local speeds and also masks remote cluster failures and network outages.

In Panache asynchronous operations consist of operations that encapsulate modifications to the cached file

[1] Currently we ignore the possibility that the mtime may not change on update. This may require content based signatures or a kernel supported change_info to verify.

system. These include relatively simple modify requests that involve a single file or directory, e.g., write, truncate, and modification of attributes such as ownership, times, and more complex requests that involve changes to the name space through updates of one or more directories, e.g., creation, deletion or renaming of a file and directory or symbolic links.

6.1 Dependent Metadata Operations

In contrast to synchronous operations, asynchronous operations modify the data and metadata at the cache cluster and then are simply queued at the gateway nodes for delayed execution at the remote cluster. Each gateway node maintains an in-memory queue of asynchronous requests that were sent by the application nodes. Each message contains the unique object identifier *fileId:* $<$ *inode_num, gen_num, fsid* $>$ of one or more objects being operated upon and the parameters of the command.

If there is a single gateway node and all the requests are queued in FIFO order, then operations will execute remotely in the same order as they did in the cache cluster. When multiple gateway nodes can push commands to the remote cluster, the distributed multi-node queue has to be controlled to maintain the desired ordering. To better understand this, let's first define some terms.

Definition 5 *A pair of update commands* $C_i(X), C_j(X)$*, on an object X, executed at the cache cluster at time* $t_i < t_j$ *are said to be* **time ordered** *, denoted by* $C_i \rightarrow C_j$*, if they need to be executed in the same relative order at the remote cluster.*

For example, commands CREATE(File_X) and WRITE(File_X, offset, length) are time ordered as the data writes cannot be pushed to the remote cluster until the file gets created.

Observation 1 *If commands* C_i, C_j, C_k *are pair-wise time ordered, i.e.,* $C_i \rightarrow C_j$ *and* $C_j \rightarrow C_k$ *then the three commands form a time ordered sequence* $C_i \rightarrow C_j \rightarrow C_k$

Definition 6 *A pair of objects* O_x, O_y*, are said to be* **dependent objects** *if there exists queued commands* C_i *and* C_j *such that* $C_i(O_x)$ *and* $C_j(O_y)$ *are time ordered.*

For example, creating a file $File_X$ and its parent directory Dir_Y make X and Y dependent objects as the parent directory create has to be pushed before the file create.

Observation 2 *If objects* O_x, O_y*, and* O_y, O_z *are pair-wise dependent, then* O_x, O_z *are also dependent objects.*

Observe that the creation of a file depends on the creation of its parent directory, which in turn depends on the creation of its parent directory, and so on. Thus, a create of a directory tree creates a chain of dependent objects. The removes follow the reverse order where the rmdir depends on the directory being empty so that the removes of the children need to execute earlier.

Definition 7 *A set of commands over a set of objects,* $C_1(O_x), C_2(O_y)...C_n(O_z)$*, are said to be* **permutable** *if they are neither time ordered nor contain dependent objects.*

Thus permutable commands can be pushed out in parallel from multiple gateway nodes without affecting correctness. For example, *create file A, create file B* are permutable among themselves.

Based on these definitions, if all commands on a given object are queued and pushed in FIFO order at the same gateway node we trivially get the time order requirements satisfied for all commands on that object. Thus, Panache hashes on the object's unique identifier, e.g., inode number and generation number, to select a gateway node on which to queue an object. It is dependent objects queued on different gateway nodes that make distributed queue ordering a challenge. To further complicate the issue, some commands such as rename and link involve multiple objects.

To maintain the distributed time ordering among dependent objects across multiple gateway node queues, we build upon the GPFS distributed token management infrastructure. This infrastructure currently coordinates access to shared objects such as inodes and byte-range locks and is explained in detail elsewhere [26]. Panache extends this distributed token infrastructure to coordinate execution of queued commands among multiple gateway nodes. The key idea is that an enqueued command acquires a shared token on objects on which it operates. Prior to the execution of a command to the remote cluster, it upgrades these tokens to exclusive, which in turn forces a token revoke on the shared tokens that are currently held by other commands on dependent objects on other gateway nodes. When a command receives a token revoke, it then also upgrades its tokens to exclusive, which results in a chain reaction of token revokes. Once a command acquires an exclusive token on its objects, it is executed and dequeued. This process results in all commands being pushed out of the distributed queues in dependent order.

The link and rename commands operate on multiple objects. Panache uses the hash function to queue these commands on multiple gateway nodes. When a multi-object request is executed, only one of the queued commands will execute to the remote cluster, with the others simply acting as placeholders to ensure intra-gateway node ordering.

6.2 Data Write Operations

On a write request, the application node first writes the data locally to the cache cluster and then sends a message to the designated gateway node to perform the write operation at the remote cluster. At a later time, the gateway node reads the data from the cache cluster and completes the remote write over pNFS.

The delayed nature of the queued write requests allow some optimizations that would not otherwise be possible if the requests had been synchronously serviced. One such optimization is write coalescing that groups the write request to match the optimal GPFS and NFS buffer sizes. The queue is also evaluated before requests are serviced to eliminate transient data updates, e.g., the creation and deletion of temporary files. All such "canceling" operations are purged without affecting the behavior of the remote cluster.

In case of remote cluster failures and network outages, all asynchronous operations can still update the cache cluster and return successfully to the application. The requests simply remain queued at the gateway nodes pending execution at the remote cluster. Any such failure, however, will affect the synchronization lag making the consistency semantics fall back to a looser eventual consistency guarantee.

6.3 Discussion

Conflict Handling Clearly, asynchronous updates can result in non-serializable executions and conflicting updates. For example, the same file may be created or updated by both the cache cluster and the remote cluster. Panache cannot prevent such conflicts, but it will detect them and resolve them based on simple policies. For example, one policy could have the cache cluster always override any conflict; another policy could move a copy of the conflicting file to a special ".conflicts" directory for manual inspection and intervention, similar to the lost+found directory generated on a normal file system check (fsck) scan. Further, it is possible to merge some types of conflicts without intervention. For example, a directory with two new files, one created by the cache and another by the remote system can be merged to form the directory containing both files. Earlier research on conflict handling of disconnected operations in Coda [25] and Intermezzo have inspired some of the techniques used in Panache after being suitably modified to handle a cluster setting.

Access control and authentication: One aspect of the caching system is that data is no more vulnerable to wrongful access as it was at the remote cluster. Panache requires userid mappings to make sure that file access permissions and ACLs setup at the remote cluster are enforced at the cache. Similarly, authentication via NFSv4's RPCSEC_GSS mechanism can be forwarded to the remote cluster to make sure end-to-end authentication can be enforced.

Recovery on Failure: The queue of pending updates can be lost due to memory pressures or a cache cluster node reboot. To avoid losing track of application updates, Panache stores sufficient persistent state to recreate the updates and synchronize the data with the remote cluster. The persistent state is stored in the inode on disk and relies on the GPFS fast inode scan to determine which inodes have been updated. Inode scans are very efficient as they can be done in parallel across multiple nodes and are basically a sequential read of the inode file. For example, in our test environment, a simple inode scan (with file attributes) on a single application node of 300K files took 2.24 seconds.

7 Evaluation

In this section we assess the performance of Panache as a scalable cache. We first use the *IOR* micro-benchmark [2] to analyze the amount of overhead Panache incurs along the data path to the remote cluster. We then use the *mdtest* micro-benchmark [4] to measure the overhead Panache incurs to queue and flush metadata operations on the gateway nodes. Finally, we run a parallel visualization application and a Hadoop application to analyze Panache with an HPC access pattern.

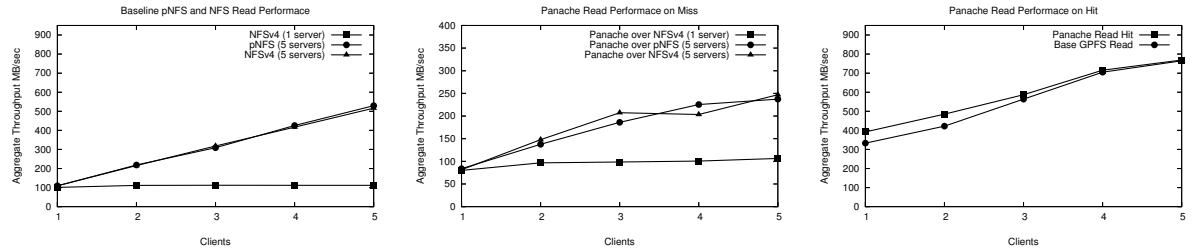
7.1 Experimental Setup

All experiments use a sixteen-node cluster connected via gigabit Ethernet, with each node assigned a different role depending on the experiment. Each node is equipped with dual 3 GHz Xeon processors, 4 GB memory and runs an experimental version of Linux 2.6.27 with pNFS. GPFS uses a 1 MB stripe size. All NFS experiments use 32 server threads and 512 KB wsize and rsize. All nodes have access to the SAN, which is comprised of a 16-port FC switch connected to a DS4800 storage controller with 12 LUNs configured for the cache cluster.

7.2 I/O Performance

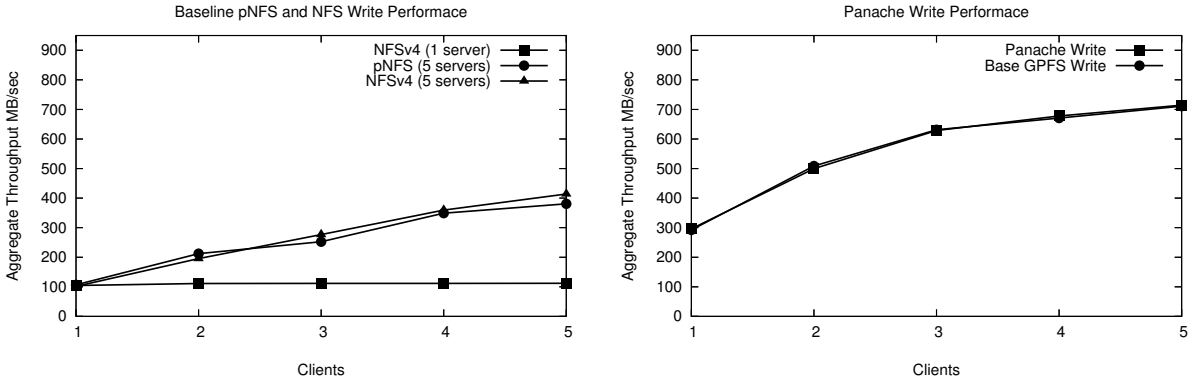
Ideally, the design of Panache is such that it should match the storage subsystem throughput on a cache hit and saturate the network bandwidth on a cache miss (assuming that the network bandwidth is less than the disk bandwidth of the cache cluster).

In the first experiment, we measure the performance reading separate 8 GB files in parallel from the remote cluster. Our local Panache cluster uses up to 5 application and gateway nodes, while the remote 5 node GPFS cluster has all nodes configured to be pNFS data servers. As we increase the number of application (client) nodes,



(a) pNFS and NFSv4 (b) Panache Cache Miss (c) Panache Cache Hit vs. Standard GPFS

Figure 5: **Aggregate Read Throughput.** *(a) pNFS and NFSv4 scale with available remote bandwidth. (b) Panache using pNFS and NFSv4 scales with available local bandwidth. (c) Panache local read performance matches standard GPFS.*



(a) pNFS and NFSv4 (b) Panache vs. Standard GPFS

Figure 6: **Aggregate Write Throughput.** *(a) pNFS and NFSv4 scale with available disk bandwidth. (b) Panache local write performance matches standard GPFS, demonstrating the negligible overhead of queuing write messages on the gateway nodes.*

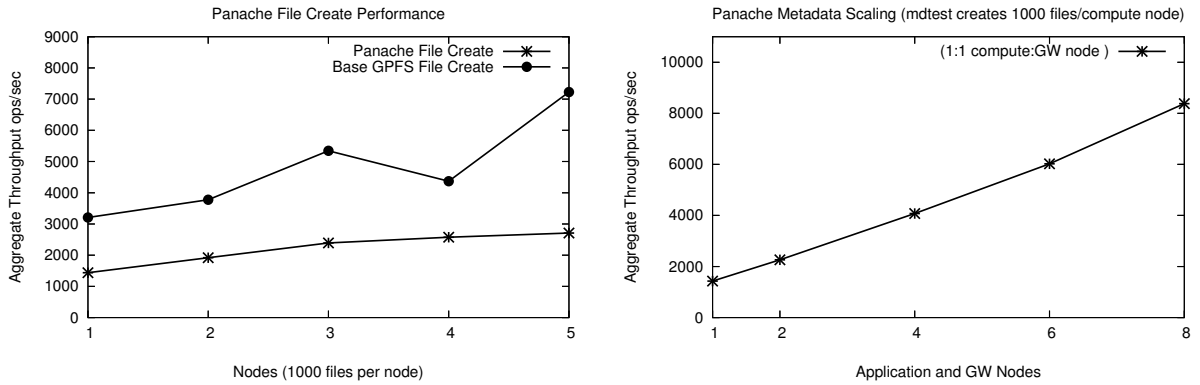
the number of gateway nodes increases as well since the miss requests are evenly dispatched. Figure 5(a) displays how the underlying data transfer mechanisms used by Panache can scale with the available bandwidth. NFSv4 with a single server is limited to the bandwidth of the single remote server while NFSv4 with multiple servers and pNFS can take advantage of all 5 available remote servers. With each NFSv4 client mounting a separate server, aggregate read throughput reaches a maximum of 516.49 MB/s with 5 clients. pNFS scales in a similar manner, reaching a maximum aggregate read throughput of 529.37 with 5 clients.

Figure 5(b) displays the aggregate read throughput of Panache utilizing pNFS and NFSv4 as its underlying transfer mechanism. The performance of Panache using NFSv4 with a single server is 5-10% less than standard NFSv4 performance. This performance hit comes from our Panache prototype, which does not fully pipeline the data between the application and gateway nodes. When Panache uses pNFS and NFSv4 using multiple servers, increasing the number of clients gives a maximum aggregate throughput of 247.16 MB/s due to a saturation of the storage network. A more robust SAN would shift the bottleneck back on the network between the local and remote clusters.

Finally, Figure 5(c) demonstrates that once a file is cached, Panache stays out of the I/O path, allowing the aggregate read throughput of Panache to match the aggregate read throughput of standard GPFS.

In the second experiment we increase the number of clients writing to a separate 8 GB files. As shown in Figure 6(b), the aggregate write throughput of Panache matches the aggregate write throughput of standard GPFS. For Panache, writes are done locally to GPFS while a write request is queued on a gateway node for asynchronous execution to the remote cluster. This experiment demonstrates that the extra step of queuing the write request on the gateway node does not impact write performance. Therefore, application write throughput is not constrained by the network bandwidth or the number of pNFS data servers, but rather by the same constraints as standard GPFS.

Eventually, data written to the cache must be synchronized to the remote cluster. Depending on the capabilities of the remote cluster, Panache can use three I/O methods: standard NFSv4 to a single server, standard NFSv4 with each client mounting a separate remote server, and pNFS. Figure 6(a) displays the ag-



(a) File metadata ops.

(b) Gateway metadata ops.

Figure 7: **Metadata performance.** *Performance of mdtest benchmark for file creates. Each node creates 1000 files in parallel. In (a), we use a single gateway node. In (b), the number of application and gateway nodes are increased in unison, with each cluster node playing both application and gateway roles.*

gregate write performance of writing separate 8 GB files to the remote cluster using these three I/O methods. Unsurprisingly, aggregate write throughput for standard NFSv4 with a single server remains flat. With each NFSv4 client mounting a separate server, aggregate write throughput reaches a maximum of 413.77 MB/s with 5 clients. pNFS scales in a similar manner, reaching a maximum aggregate write throughput of 380.78 MB/s with 5 clients. Neither NFSv4 with multiple servers nor pNFS saturate the available network bandwidth due to limitations in the disk subsystem.

It is important to note that although the performance of pNFS and NFSv4 with multiple servers appears on the surface to be similar, the lack of coordinated access in NFSv4 creates several performance hurdles. For instance, if there are a greater number of gateway nodes than remote servers, NFSv4 clients will not be evenly load balanced among the servers, creating possible hot spots. pNFS avoids this by always balancing I/O requests among the remote servers evenly. In addition, NFSv4 unaligned file writes across multiple servers can create false sharing of data blocks, causing the cluster file system to lock and flush data unnecessarily.

7.3 Metadata Performance

To measure the metadata update performance in the cache cluster we use the *mdtest* benchmark, which performs file creates from multiple nodes in the cluster. Figure 7(a) shows the aggregate throughput of 1000 file create operations per cluster node. With 4 application nodes simultaneously creating a total of 4000 files, the Panache throughput (2574 ops/s) is roughly half that of the local GPFS (4370 ops/s) performance. The Panache code path has the added overhead of first creating the file locally and then sending a RPC to queue the operation on a gateway node. As the graph shows, as the

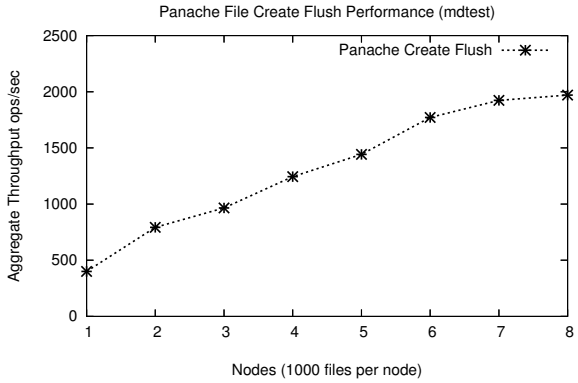


Figure 8: **Metadata flush performance.** *Performance of mdtest benchmark for file creates with flush. Each node flushes 1000 files in parallel back to the home cluster.*

number of nodes increases, we can saturate the single gateway node. To see the impact of increasing the number of gateway nodes, Figure 7(b) demonstrates the scale up when the number of application nodes and gateway nodes increase in tandem, up to a maximum of 8 cache and remote nodes.

As all updates are asynchronous, we also demonstrate the performance of flushing file creates to the remote cluster in Figure 8. By increasing the number of gateway and remote nodes in tandem, we can scale the number of creates per second from 400 to 2000, a five fold increase for 7 additional nodes. The lack of linear increase is due to our prototype's inefficient use of the GPFS token management service.

7.4 WAN Performance

To validate the effectiveness of Panache over a WAN we used the IOR parallel file read benchmark and the Linux tc command. The WAN represented the 30ms la-

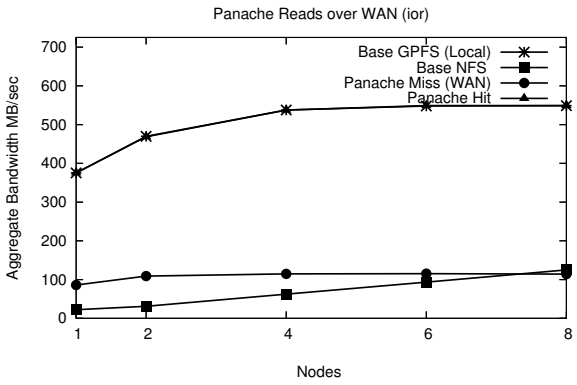


Figure 9: **IOR file Reads over a WAN.** *The 8 node cache cluster and 8 node remote cluster are separated by a 30ms latency link. Each file is 5GB in size.*

tency link between the IBM San Jose and Tucson facilities. The cache and remote clusters both contain 8 nodes, keeping the gateway and remote nodes in tandem. Figure 9 shows the aggregate bandwidth on both a hit and a miss for an increasing number of nodes in the cluster. The hit bandwidth matches that of a local GPFS read. For cache miss, while Panache can utilize parallel ingest to increase performance initially, both Panache and NFS eventually suffer from slow network bandwidth.

7.5 Visualization for Cognitive Models

This section evaluates Panache with a real supercomputing application that visualizes the $8x10^6$ neural firings of a large scale cognitive model of a mouse brain [23]. The cognitive model runs at a remote cluster (a BlueGene/L system with 4096 nodes) and the visualization application runs at the cache cluster and creates a "movie" as output. In the experiment in Table 3, we copied a fraction of the data (64 files of 200MB each) generated by the cognitive model to our 5 node remote cluster and ran the visualization application on the Panache cluster. The application reads in the data and creates a movie file of 250MB. Visualization is a CPU-bound operation, but asynchronous writes helped Panache reduce runtime over pNFS by 14 percent. Once the data is cached, time to regenerate the visualization files is reduced by an additional 17.6 percent.

pNFS	Panache (miss)	Panache (hit)
46.74 (s)	40.2 (s)	31.96 (s)

Table 3: **Supercomputing application.** *pNFS includes remote cluster reads and writes. Panache Miss reads from the remote and asynchronous write back. Panache Hit reads from the cache and asynchronous write back.*

7.6 MapReduce Application

The MapReduce framework provides a programmable infrastructure to build highly parallel applications that operate on large data sets [11]. Using this framework, applications define a map function that defines a key and operates on a chunk of the data. The reduce function aggregates the results for a given key. Developers may write several MapReduce programs to extract different properties from a single data set, building a use case for remote caching. We use the MapReduce framework from Hadoop 0.20.1 [6] and configured it to use Panache as the underlying distributed store (instead of the HDFS file system it uses by default).

Table 4 presents the performance of Distributed Grep, a canonical MapReduce example application, over a data set of 16 files, 500MB each, running in parallel across 8 nodes with the remote cluster also consisting of 8 nodes. The GPFS result was the baseline result where the data was already available in the local GPFS cluster. In the Panache miss case, as the distributed grep application accessed the input files, the gateway nodes dynamically ingested the data in parallel from the remote cluster. In the hit case, Panache revalidated the data every 15 secs with the remote cluster. This experiment validates our assertion that data can be dynamically cached and immediately available for parallel access from multiple nodes within the cluster.

Hadoop+GPFS	Hadoop+Panache		
Local	Miss LAN	Miss WAN	Hit
81.6 (s)	113.1 (s)	140.6 (s)	86.5 (s)

Table 4: **MapReduce application.** *Distributed Grep using the Hadoop framework over GPFS and Panache. The WAN results are over a 30ms latency link.*

8 Related Work

Distributed file systems have been an active area of research for almost two decades. NFS is among the most widely-used distributed networked file systems. Other variants of NFS, Spritely NFS [28] and NQNFS [20] added stronger consistency semantics to NFS by adding server callbacks and leases. NFSv4 greatly enhances wide-area access support, optimizes consistency support via delegations, and improves compatibility with Windows. The latest revision, NFSv4.1, also adds parallel data access across a variety of clustered file and storage systems. In the non-Unix world, the Common Internet File System (CIFS) protocol is used to allow MS-Windows hosts to share data over the Internet. While these distributed file systems provide remote file access and some limited in-memory client caching they cannot operate across multiple nodes and in the presence of net-

work and server failures.

Apart from NFS, another widely studied globally distributed file system is AFS [17]. It provides close-to-open consistency, supports client-side persistent caching, and relies on client callbacks as the primary mechanism for cache revalidation. Later, Coda [25] and Ficus [24] dealt with replication for better scalability while focusing on disconnected operations for greater data availability in the event of a network partition.

More recently, the work on TierStore applies some of the same principles for the development and deployment of applications in bandwidth challenged networks [13]. It defines Delay Tolerant Networking with a store-and-forward network overlay and a publish/subscribe-based multicast replication protocol. In limited bandwidth environments, LBFS takes a different approach by focusing on reducing bandwidth usage by eliminating cross-file similarities [22]. Panache can easily absorb some of its similarity techniques to reduce the data transfer to and from the cache.

A plethora of commercial WAFS and WAN acceleration products provide caching for NFS and CIFS using custom devices and proprietary protocols [1]. Panache differs from WAFS solutions as it relies on standard protocols between the remote and cache sites. Muntz and Honeyman [21] looked at multi-level caching to solve scaling problems in distributed file systems but questioned its effectiveness. However, their observations may not hold today as the advances in network bandwidth, web-based applications, and the emerging trends of cloud stores have substantially increased remote collaboration. Furthermore, cooperative caching, both in the web and file system space, has been extensively studied [10]. The primary focus, however, has been to expand the cache space available by sharing data across sites to improve hit rates.

Lustre [3] and PanFS [29] are highly-scalable object based cluster file systems. These efforts have focused on improving file-serving performance and are not designed for remotely accessing data from existing file servers and NAS appliances over a WAN.

FS-Cache is a single-node caching file system layer for Linux that can be used to enhance the performance of a distributed file system such as NFS [18]. FS-Cache is not a standalone file system; instead it is meant to work with the front and back file systems. Unlike Panache, it does not mimic the namespace of the remote file system and does not provide direct POSIX access to the cache. Moreover, FS-Cache is a single node system and is not designed for multiple nodes of a cluster accessing the cache concurrently. Similar implementations such as CacheFS are available on other platforms such as Solaris and as a stackable file system with improved cache policies [27].

A number of research efforts have focused on building large scale distributed storage facilities using customized protocols and replication. The Bayou [12] project introduced eventual consistency across replicas, an idea that we borrowed in Panache for converging to a consistent state after failure. The Oceanstore project [19] used Byzantine agreement techniques to coordinate access between the primary replica and the secondaries. The PRACTI replication framework [9] separated the flow of cache invalidation traffic from that of data itself. Others like Farsite [8] enabled unreliable servers to combine their resources into a highly-available and reliable file storage facility.

Recently the success of file sharing on the Web, especially BitTorrent [5] which has been widely studied, has triggered renewed effort for applying similar ideas to build peer-to-peer storage systems. BitTorrent's chunk-based data retrieval method that enables clients to fetch data in parallel from multiple remote sources is similar to the implementation of parallel reads in Panache.

9 Conclusions

This paper introduced Panache, a scalable, high-performance, clustered file system cache that promises seamless access to massive and remote datasets. Panache supports a POSIX interface and employs a fully parallelizable design, enabling applications to saturate available network and compute hardware. Panache can also mask fluctuating WAN latencies and outages by acting as a standalone file system under adverse conditions.

We evaluated Panache using several data and metadata micro-benchmarks in local and wide area networks, demonstrating the scalability of using multiple gateway nodes to flush and ingest data from a remote cluster. We also demonstrated the benefits for both a visualization and analytics application. As Panache achieves the performance of a clustered file system on a cache hit, large scale applications can leverage a clustered caching solution without paying the performance penalty of accessing remote data using out-of-band techniques.

References

[1] Blue Coat Systems, Inc. `www.bluecoat.com`.

[2] IOR Benchmark. `sourceforge.net/projects/ior-sio`.

[3] Lustre file system. `www.lustre.org`.

[4] Mdtest benchmark. `sourceforge.net/projects/mdtest`.

[5] Bittorrent. `www.bittorrent.com`.

[6] Hadoop Distribued Filesystem. `hadoop.apache.org`.

[7] Nirvanix Storage Delivery Network. `www.nirvanix.com`.

[8] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch,

M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. of the 4th Symposium on Operating Systems Design and Implementation*, 2002.

[9] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *Proc. of the 3rd USENIX Symposium on Networked Systems Design and Implementation*, 2006.

[10] M. Dahlin, R. Wang, T. E. Anderson, and D. A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proc. of the 1st Symposium on Operating Systems Design and Implementation*, 1994.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. of the 6th Symposium on Operating System Design and Implementation*, 2004.

[12] A. J. Demers, K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and B. B. Welch. The Bayou architecture: Support for data sharing among mobile users. In *Proc. of the IEEE Workshop on Mobile Computing Systems & Applications*, 1994.

[13] M. Demmer, B. Du, and E. Brewer. Tierstore: a distributed filesystem for challenged networks in developing regions. In *Proc. of the 6th USENIX Conference on File and Storage Technologies*, 2008.

[14] S. Ghemawat, H. Gobioff, and S. Leung. The google file system. In *Proc. of the 19th ACM symposium on operating systems principles*, 2003.

[15] A. Gulati, M. Naik, and R. Tewari. Nache: Design and Implementation of a Caching Proxy for NFSv4. In *Proc. of the Fifth Conference on File and Storage Technologies*, 2007.

[16] D. Hildebrand and P. Honeyman. Exporting storage systems in a scalable manner with pNFS. In *Proc. of the 22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies*, 2005.

[17] 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(1):51–81, 1988.

[18] D. Howells. FS-Cache: A Network Filesystem Caching Facility. In *Proc. of the Linux Symposium*, 2006.

[19] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[20] R. Macklem. Not Quite NFS, soft cache consistency for NFS. In *Proc. of the USENIX Winter Technical Conference*, 1994.

[21] D. Muntz and P. Honeyman. Multi-level Caching in Distributed File Systems. In *Proc. of the USENIX Winter Technical Conference*, 1992.

[22] A. Muthitacharoen, B. Chen, and D. Mazi. A low-bandwidth network file system. In *Proc. of the 18th ACM symposium on operating systems principles*, 2001.

[23] A. Rajagopal and D. Modha. Anatomy of a cortical simulator. In *Proc. of Supercomputing '07*, 2007.

[24] P. Reiher, J. Heidemann, D. Ratner, G. Skinner, and G. Popek. Resolving file conflicts in the Ficus file system. In *Proc. of the USENIX Summer Technical Conference*, 1994.

[25] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[26] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proc. of the First Conference on File and Storage Technologies*, 2002.

[27] G. Sivathanu and E. Zadok. A Versatile Persistent Caching Framework for File Systems. *Technical Report FSL-05-05, Stony Brook University*, 2005.

[28] V. Srinivasan and J. Mogul. Spritely NFS: experiments with cache-consistency protocols. In *Proc. of the 12th Symposium on Operating Systems Principles*, 1989.

[29] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable Performance of the Panasas Parallel File System. In *Proc. of the 6th Conference on File and Storage Technologies*, 2008.

BASIL: Automated IO Load Balancing Across Storage Devices

Ajay Gulati
VMware, Inc.
agulati@vmware.com

Chethan Kumar
VMware, Inc.
ckumar@vmware.com

Irfan Ahmad
VMware, Inc.
irfan@vmware.com

Karan Kumar
Carnegie Mellon University
karank@andrew.cmu.edu

Abstract

Live migration of virtual hard disks between storage arrays has long been possible. However, there is a dearth of online tools to perform automated virtual disk placement and IO load balancing across multiple storage arrays. This problem is quite challenging because the performance of IO workloads depends heavily on their own characteristics and that of the underlying storage device. Moreover, many device-specific details are hidden behind the interface exposed by storage arrays.

In this paper, we introduce BASIL, a novel software system that automatically manages virtual disk placement and performs load balancing across devices without assuming any support from the storage arrays. BASIL uses IO latency as a primary metric for modeling. Our technique involves separate online modeling of workloads and storage devices. BASIL uses these models to recommend migrations between devices to balance load and improve overall performance.

We present the design and implementation of BASIL in the context of VMware ESX, a hypervisor-based virtualization system, and demonstrate that the modeling works well for a wide range of workloads and devices. We evaluate the placements recommended by BASIL, and show that they lead to improvements of at least 25% in both latency and throughput for 80 percent of the hundreds of microbenchmark configurations we ran. When tested with enterprise applications, BASIL performed favorably versus human experts, improving latency by 18-27%.

1 Introduction

Live migration of virtual machines has been used extensively in order to manage CPU and memory resources, and to improve overall utilization across multiple physical hosts. Tools such as VMware's Distributed Resource Scheduler (DRS) perform automated placement of virtual machines (VMs) on a cluster of hosts in an efficient and effective manner [6]. However, automatic placement and load balancing of IO workloads across a set of storage devices has remained an open problem. Diverse IO behavior from various workloads and hot-spotting can cause significant imbalance across devices over time.

An automated tool would also enable the aggregation of multiple storage devices (LUNs), also known as data stores, into a single, flexible pool of storage that we call a POD (*i.e.* Pool of Data stores). Administrators can dynamically populate PODs with data stores of similar reliability characteristics and then just associate virtual disks with a POD. The load balancer would take care of initial placement as well as future migrations based on actual workload measurements. The flexibility of separating the physical from the logical greatly simplifies storage management by allowing data stores to be efficiently and dynamically added or removed from PODs to deal with maintenance, out of space conditions and performance issues.

In spite of significant research towards storage configuration, workload characterization, array modeling and automatic data placement [8, 10, 12, 15, 21], most storage administrators in IT organizations today rely on rules of thumb and ad hoc techniques, both for configuring a storage array and laying out data on different LUNs. For example, placement of workloads is often based on balancing space consumption or the number of workloads on each data store, which can lead to hot-spotting of IOs on fewer devices. Over-provisioning is also used in some cases to mitigate real or perceived performance issues and to isolate top-tier workloads.

The need for a storage management utility is even greater in virtualized environments because of high degrees of storage consolidation and sprawl of virtual disks over tens to hundreds of data stores. Figure 1 shows a typical setup in a virtualized datacenter, where a set of hosts has access to multiple shared data stores. The storage array is carved up into groups of disks with some RAID level configuration. Each such disk group is further di-

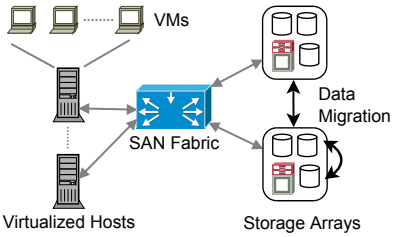


Figure 1: Live virtual disk migration between devices.

vided into LUNs which are exported to hosts as storage devices (referred to interchangeably as data stores). Initial placement of virtual disks and data migration across different data stores should be guided by workload characterization, device modeling and analysis to improve IO performance as well as utilization of storage devices. This is more difficult than CPU or memory allocation because storage is a stateful resource: IO performance depends strongly on workload and device characteristics.

In this paper, we present the design and implementation of BASIL, a light-weight online storage management system. BASIL is novel in two key ways: (1) identifying IO latency as the primary metric for modeling, and (2) using simple models both for workloads and devices that can be obtained efficiently online. BASIL uses IO latency as the main metric because of its near linear relationship with application-level characteristics (shown later in Section 3). Throughput and bandwidth, on the other hand, behave non-linearly with respect to various workload characteristics.

For modeling, we partition the measurements into two sets. First are the properties that are inherent to a workload and mostly independent of the underlying device such as seek-distance profile, IO size, read-write ratio and number of outstanding IOs. Second are device dependent measurements such as IOPS and IO latency. We use the first set to model workloads and a subset of the latter to model devices. Based on measurements and the corresponding models, the analyzer assigns the IO load in proportion to the performance of each storage device.

We have prototyped BASIL in a real environment with a set of virtualized servers, each running multiple VMs placed across many data stores. Our extensive evaluation based on hundreds of workloads and tens of device configurations shows that our models are simple yet effective. Results indicate that BASIL achieves improvements in throughput of at least 25% and latency reduction of at least 33% in over 80 percent of all of our test configurations. In fact, approximately half the tests cases saw at least 50% better throughput and latency. BASIL achieves optimal initial placement of virtual disks in 68% of our experiments. For load balancing of enterprise applications, BASIL outperforms human experts by improving latency by 18-27% and throughput by up to 10%.

The next section presents some background on the relevant prior work and a comparison with BASIL. Section 3 discusses details of our workload characterization and modeling techniques. Device modeling techniques and storage specific issues are discussed in Section 4. Load balancing and initial placement algorithms are described in Section 5. Section 6 presents the results of our extensive evaluation on real testbeds. Finally, we conclude with some directions for future work in Section 7.

2 Background and Prior Art

Storage management has been an active area of research in the past decade but the state of the art still consists of rules of thumb, guess work and extensive manual tuning. Prior work has focused on a variety of related problems such as disk drive and array modeling, storage array configuration, workload characterization and data migration.

Existing modeling approaches can be classified as either *white-box* or *black-box*, based on the need for detailed information about internals of a storage device. Black-box models are generally preferred because they are oblivious to the internal details of arrays and can be widely deployed in practice. Another classification is based on *absolute* vs. *relative* modeling of devices. Absolute models try to predict the actual bandwidth, IOPS and/or latency for a given workload when placed on a storage device. In contrast, a relative model may just provide the relative change in performance of a workload from device A to B. The latter is more useful if a workload's performance on one of the devices is already known. Our approach (BASIL) is a black-box technique that relies on the relative performance modeling of storage devices.

Automated management tools such as Hippodrome [10] and Minerva [8] have been proposed in prior work to ease the tasks of a storage administrator. Hippodrome automates storage system configuration by iterating over three stages: analyze workloads, design the new system and implement the new design. Similarly, Minerva [8] uses a declarative specification of application requirements and device capabilities to solve a constraint-based optimization problem for storage-system design. The goal is to come up with the best array configuration for a workload. The workload characteristics used by both Minerva and Hippodrome are somewhat more detailed and different than ours. These tools are trying to solve a different and a more difficult problem of optimizing overall storage system configuration. We instead focus on load balancing of IO workloads among existing storage devices across multiple arrays.

Mesnier *et al.* [15] proposed a black-box approach based on evaluating relative fitness of storage devices to predict the performance of a workload as it is moved

from its current storage device to another. Their approach requires extensive training data to create relative fitness models among every *pair* of devices. Practically speaking, this is hard to do in an enterprise environment where storage devices may get added over time and may not be available for such analysis. They also do very extensive offline modeling for bandwidth, IOPS and latency and we derive a much simpler device model consisting of a single parameter in a completely online manner. As such, our models may be somewhat less detailed or less accurate, but experimentation shows that they work well enough in practice to guide our load balancer. Their model can potentially be integrated with our load balancer as an input into our own device modeling.

Analytical models have been proposed in the past for both single disk drives and storage arrays [14, 17, 19, 20]. Other models include table-based [9] and machine learning [22] techniques. These models try to accurately predict the performance of a storage device given a particular workload. Most analytical models require detailed knowledge of the storage device such as sectors per track, cache sizes, read-ahead policies, RAID type, RPM for disks etc. Such information is very hard to obtain automatically in real systems, and most of it is abstracted out in the interfaces presented by storage arrays to the hosts. Others need an extensive offline analysis to generate device models. One key requirement that BASIL addresses is using only the information that can be easily collected online in a live system using existing performance monitoring tools. While one can clearly make better predictions given more detailed information and exclusive, offline access to storage devices, we don't consider this practical for real deployments.

3 Workload Characterization

Any attempt at designing intelligent IO-aware placement policies must start with storage workload characterization as an essential first step. For each workload in our system, we currently track the average IO latency along the following parameters: seek distance, IO sizes, read-write ratio and average number of outstanding IOs. We use the VMware ESX hypervisor, in which these parameters can be easily obtained for each VM and each virtual disk in an online, light-weight and transparent manner [7]. A similar tool is available for Xen [18]. Data is collected for both reads and writes to identify any potential anomalies in the application or device behavior towards different request types.

We have observed that, to the first approximation, four of our measured parameters (*i.e.*, randomness, IO size, read-write ratio and average outstanding IOs) are inherent to a workload and are mostly independent of the underlying device. In actual fact, some of the characteristics that we classify as inherent to a workload can indeed be partially dependent on the response times delivered by the storage device; *e.g.*, IO sizes for a database logger might decrease as IO latencies decrease. In previous work [15], Mesnier *et al.* modeled the change in workload as it is moved from one device to another. According to their data, most characteristics showed a small change except write seek distance. Our model makes this assumption for simplicity and errors associated with this assumption appear to be quite small.

Our workload model tries to predict a notion of load that a workload might induce on storage devices using these characteristics. In order to develop a model, we ran a large set of experiments varying the values of each of these parameters using Iometer [3] inside a Microsoft Windows 2003 VM accessing a 4-disk RAID-0 LUN on an EMC CLARiiON array. The set of values chosen for our 750 configurations are a cross-product of:

Outstanding IOs	{4, 8, 16, 32, 64}
IO size (in KB)	{8, 16, 32, 128, 256, 512}
Read%	{0, 25, 50, 75, 100}
Random%	{0, 25, 50, 75, 100}

For each of these configurations we obtain the values of average IO latency and IOPS, both for reads and writes. For the purpose of workload modeling, we next discuss some representative sample observations of average IO latency for each one of these parameters while keeping the others fixed. Figure 2(a) shows the relationship between IO latency and outstanding IOs (OIOs) for various workload configurations. We note that latency varies linearly with the number of outstanding IOs for all the configurations. This is expected because as the total number of OIOs increases, the overall queuing delay should increase linearly with it. For very small number of OIOs, we may see non-linear behavior because of the improvement in device throughput but over a reasonable range (8-64) of OIOs, we consistently observe very linear behavior. Similarly, IO latency tends to vary linearly with the variation in IO sizes as shown in Figure 2(b). This is because the transmission delay increases linearly with IO size.

Figure 2(c) shows the variation of IO latency as we increase the percentage of reads in the workload. Interestingly, the latency again varies linearly with read percentage except for some non-linearity around corner cases such as completely sequential workloads. We use the read-write ratio as a parameter in our modeling because we noticed that, for most cases, the read latencies were very different compared to write (almost an order of magnitude higher) making it important to characterize a workload using this parameter. We believe that the difference in latencies is mainly due to the fact that writes return once they are written to the cache at the array and the latency of destaging is hidden from the application. Of course, in cases where the cache is almost full, the

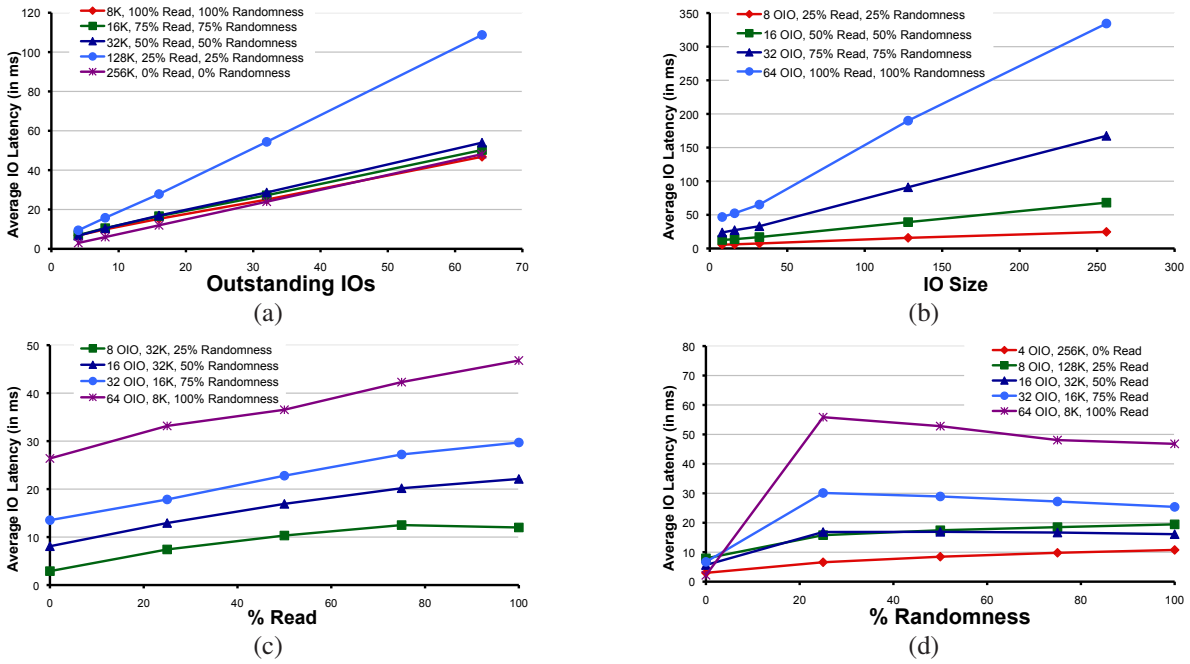


Figure 2: Variation of IO latency with respect to each of the four workload characteristics: outstanding IOs, IO size, % Reads and % Randomness. Experiments run on a 4-disk RAID-0 LUN on an EMC CLARiiON CX3-40 array.

writes may see latencies closer to the reads. We believe this to be fairly uncommon especially given the burstiness of most enterprise applications [12]. Finally, the variation of latency with random% is shown in Figure 2(d). Notice the linear relationship with a very small slope, except for a big drop in latency for the completely sequential workload. These results show that except for extreme cases such as 100% sequential or 100% write workloads, the behavior of latency with respect to these parameters is quite close to linear[1]. Another key observation is that the cases where we typically observe non-linearity are easy to identify using their online characterization.

Based on these observations, we modeled the IO latency (L) of a workload using the following equation:

$$L = \frac{(K_1 + OIO)(K_2 + IOsize)(K_3 + \frac{read\%}{100})(K_4 + \frac{random\%}{100})}{K_5} \quad (1)$$

We compute all of the constants in the above equation using the data points available to us. We explain the computation of K_1 here, other constants K_2, K_3 and K_4 are computed in a similar manner. To compute K_1, we take two latency measurements with different OIO values but the same value for the other three workload parameters. Then by dividing the two equations we get:

$$\frac{L_1}{L_2} = \frac{K_1 + OIO_1}{K_1 + OIO_2} \quad (2)$$

[1]The small negative slope in some cases in Figure 2(d) with large OIOs is due to known prefetching issues in our target array's firmware version. This effect went away when prefetching is turned off.

$$K_1 = \frac{OIO_1 - OIO_2 * L_1/L_2}{L_1/L_2 - 1} \quad (3)$$

We compute the value of K_1 for all pairs where the three parameters except OIO are identical and take the median of the set of values obtained as K_1. The values of K_1 fall within a range with some outliers and picking a median ensures that we are not biased by a few extreme values. We repeat the same procedure to obtain other constants in the numerator of Equation 1.

To obtain the value of K_5, we compute a linear fit between actual latency values and the value of the numerator based on K_i values. Linear fitting returns the value of K_5 that minimizes the least square error between the actual measured values of latency and our estimated values.

Using IO latencies for training our workload model creates some dependence on the underlying device and storage array architectures. While this isn't ideal, we argue that as a practical matter, if the associated errors are small enough, and if the high error cases can usually be identified and dealt with separately, the simplicity of our modeling approach makes it an attractive technique.

Once we determined all the constants of the model in Equation 1, we compared the computed and actual latency values. Figure 3(a) (LUN1) shows the relative error between the actual and computed latency values for all workload configurations. Note that the computed values do a fairly good job of tracking the actual values in most cases. We individually studied the data points with high errors and the majority of those were sequential IO

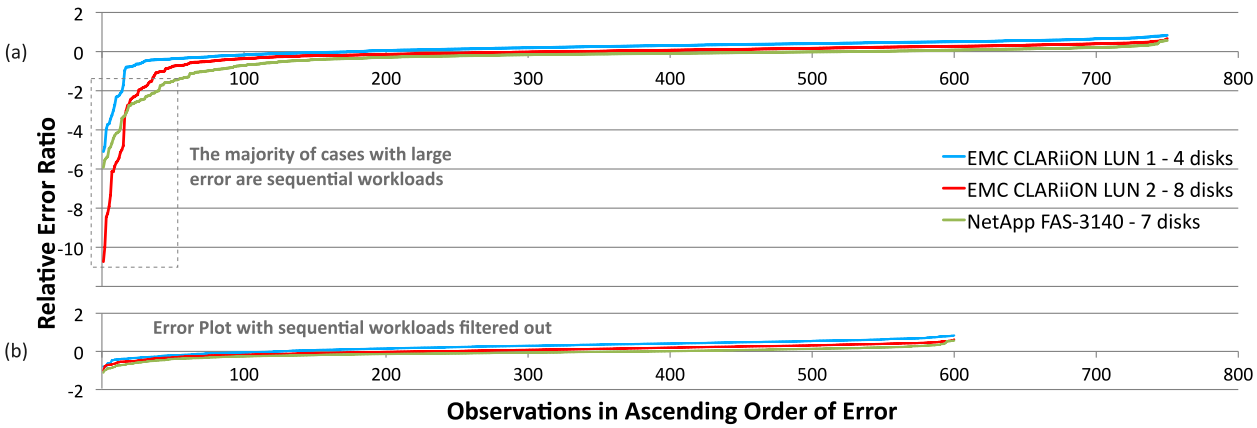


Figure 3: Relative error in latency computation based on our formula and actual latency values observed.

or write-only patterns. Figure 3(b) plots the same data but with the 100% sequential workloads filtered out.

In order to validate our modeling technique, we ran the same 750 workload configurations on a different LUN on the same EMC storage array, this time with 8 disks. We used the same values of K_1, K_2, K_3 and K_4 as computed before on the 4-disk LUN. Since the disk types and RAID configuration was identical, K_5 should vary in proportion with the number of disks, so we doubled the value, as the number of disks is doubled in this case. Figure 3 (LUN 2) again shows the error between actual and computed latency values for various workload configurations. Note that the computed values based on the previous constants are fairly good at tracking the actual values. We again noticed that most of the high error cases were due to the poor prediction for corner cases, such as 100% sequential, 100% writes, etc.

To understand variation across different storage architectures, we ran a similar set of 750 tests on a NetApp FAS-3140 storage array. The experiments were run on a 256 GB virtual disk created on a 500 GB LUN backed by a 7-disk RAID-6 (double parity) group. Figures 4(a), (b), (c) and (d) show the relationship between average IO latency with OIOs, IO size, Read% and Random% respectively. Again for OIOs, IO size and Random%, we observed a linear behavior with positive slope. However, for the Read% case on the NetApp array, the slope was close to zero or slightly negative. We also found that the read latencies were very close to or slightly smaller than write latencies in most cases. We believe this is due to a small NVRAM cache in the array (512 MB). The writes are getting flushed to the disks in a synchronous manner and array is giving slight preference to reads over writes. We again modeled the system using Equation 1, calculated the K_i constants and computed the relative error in the measured and computed latencies using the NetApp measurements. Figure 3 (NetApp) shows the relative error for all 750 cases. We looked into the mapping of cases with high error with the actual configurations and noticed that almost all of those configurations are completely sequential workloads. This shows that our linear model over-predicts the latency for 100% sequential workloads because the linearity assumption doesn't hold in such extreme cases. Figures 2(d) and 4(d) also show a big drop in latency as we go from 25% random to 0% random. We looked at the relationship between IO latency and workload parameters for such extreme cases. Figure 5 shows that for sequential cases the relationship between IO latency and read% is not quite linear.

In practice, we think such cases are less common and poor prediction for such cases is not as critical. Earlier work in the area of workload characterization [12,13] confirms our experience. Most enterprise and web workloads that have been studied including Microsoft Exchange, a maps server, and TPC-C and TPC-E like workloads exhibit very little sequential accesses. The only notable workloads that have greater than 75% sequentiality are decision support systems.

Since K_5 is a device dependent parameter, we use the numerator of Equation 1 to represent the load metric ($\mathscr{L}$) for a workload. Based on our experience and empirical data, K_1, K_2, K_3 and K_4 lie in a narrow range even when measured across devices. This gives us a choice when applying our modeling on a real system: we can use a fixed set of values for the constants or recalibrate the model by computing the constants on a per-device basis in an offline manner when a device is first provisioned and added to the storage POD.

4 Storage Device Modeling

So far we have discussed the modeling of workloads based on the parameters that are inherent to a workload. In this section we present our device modeling technique using the measurements dependent on the performance of the device. Most of the device-level characteristics such

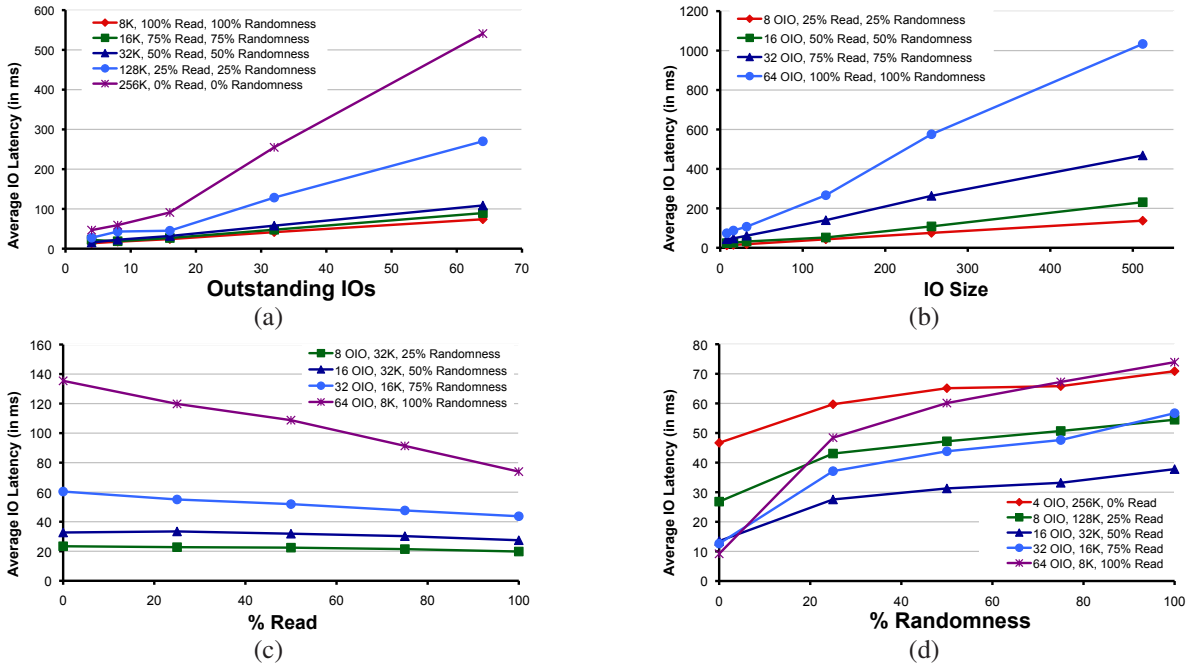


Figure 4: Variation of IO latency with respect to each of the four workload characteristics: outstanding IOs, IO size, % Reads and % Randomness. Experiments run on a 7-disk RAID-6 LUN on a NetApp FAS-3140 array.

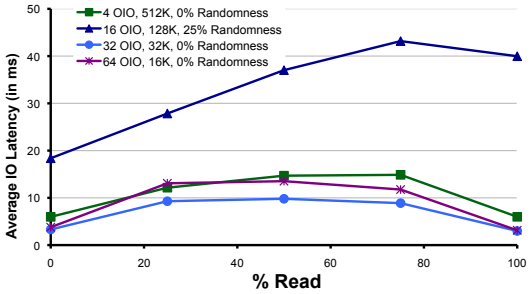


Figure 5: Varying Read% for the Anomalous Workloads

as number of disk spindles backing a LUN, disk-level features such as RPM, average seek delay, etc. are hidden from the hosts. Storage arrays only expose a LUN as a logical device. This makes it very hard to make load balancing decisions because we don't know if a workload is being moved from a LUN with 20 disks to a LUN with 5 disks, or from a LUN with faster Fibre Channel (FC) disk drives to a LUN with slower SATA drives.

For device modeling, instead of trying to obtain a white-box model of the LUNs, we use IO latency as the main performance metric. We collect information pairs consisting of number of outstanding IOs and average IO latency observed. In any time interval, hosts know the average number of outstanding IOs that are sent to a LUN and they also measure the average IO latency observed by the IOs. This information can be easily gathered using existing tools such as esxtop or xentop, without any extra overhead. For clustered environments, where multiple hosts access the same LUN, we aggregate this information across hosts to get a complete view.

We have observed that IO latency increases linearly with the increase in number of outstanding IOs (*i.e.*, load) on the array. This is also shown in earlier studies [11]. Given this knowledge, we use the set of data points of the form $\langle OIO, Latency \rangle$ over a period of time and compute a linear fit which minimizes the least squares error for the data points. The slope of the resulting line would indicate the overall performance capability of the LUN. We believe that this should cover cases where LUNs have different number of disks and where disks have diverse characteristics, *e.g.*, enterprise-class FC vs SATA disks.

We conducted a simple experiment using LUNs with different number of disks and measured the slope of the linear fit line. An illustrative workload of 8KB random IOs is run on each of the LUNs using a Windows 2003 VM running Iometer [3]. Figure 6 shows the variation of IO latency with OIOs for LUNs with 4 to 16 disks. Note that the slopes vary inversely with the number of disks.

To understand the behavior in presence of different disk types, we ran an experiment on a NetApp FAS-3140 storage array using two LUNs, each with seven disks and dual parity RAID. LUN1 consisted of enterprise class FC disks (134 GB each) and LUN2 consisted of slower SATA disks (414 GB each). We created virtual disks of size 256 GB on each of the LUNs and ran a workload

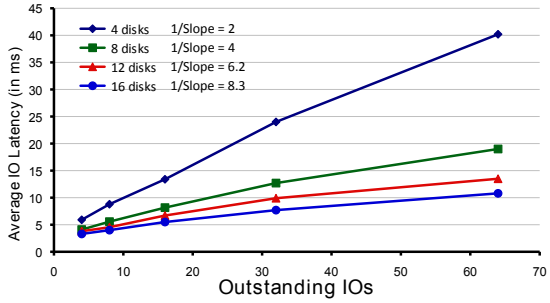


Figure 6: Device Modeling: different number of disks

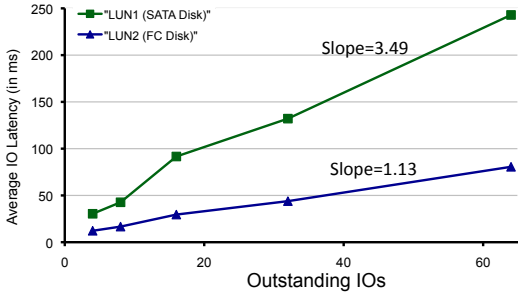


Figure 7: Device Modeling: different disk types

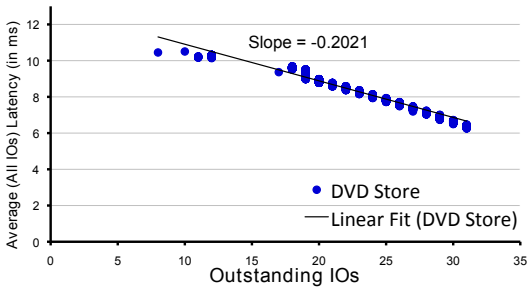


Figure 8: Negative slope in case of running DVD Store workload on a LUN. This happens due to a large number of writes happening during periods of high OIOs.

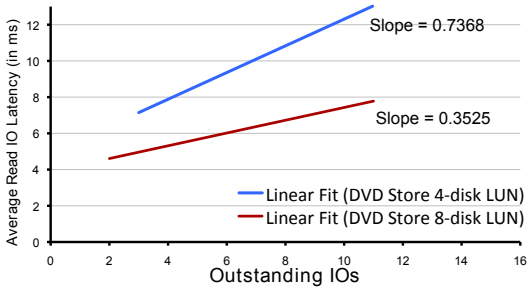


Figure 9: This plot shows the slopes for two data stores, both running DVD Store. Writes are filtered out in the model. The slopes are positive here and the slope value is lower for the 8 disk LUN.

with 80% reads, 70% randomness and 16KB IOs, with different values of OIOs. The workloads were generated using Iometer [3] inside a Windows 2003 VM. Figure 7 shows the average latency observed for these two LUNs with respect to OIOs. Note that the slope for LUN1 with faster disks is 1.13, which is lower compared to the slope of 3.5 for LUN2 with slower disks.

This data shows that the performance of a LUN can be estimated by looking at the slope of relationship between average latency and outstanding IOs over a long time interval. Based on these results, we define a performance parameter $\mathscr{P}$ to be the inverse of the slope obtained by computing a linear fit on the $\langle OIO, Latency \rangle$ data pairs collected for that LUN.

4.1 Storage-specific Challenges

Storage devices are stateful, and IO latencies observed are dependent on the actual workload going to the LUN. For example, writes and sequential IOs may have very different latencies compared to reads and random IOs, respectively. This can create problems for device modeling if the IO behavior is different for various OIO values. We observed this behavior while experimenting with the DVD Store [1] database test suite, which represents a complete online e-commerce application running on SQL databases. The setup consisted of one database LUN and one log LUN, of sizes 250 GB and 10 GB respectively. Figure 8 shows the distribution of OIO and latency pairs for a 30 minute run of DVD Store. Note that the slope turned out to be slightly negative, which is not desirable for modeling. Upon investigation, we found that the data points with larger OIO values were bursty writes that have smaller latencies because of write caching at the array.

Similar anomalies can happen for other cases: (1) Sequential IOs: the slope can be negative if IOs are highly sequential during the periods of large OIOs and random for smaller OIO values. (2) Large IO sizes: the slope can be negative if the IO sizes are large during the period of low OIOs and small during high OIO periods. All these workload-specific details and extreme cases can adversely impact the workload model.

In order to mitigate this issue, we made two modifications to our model: first, we consider only read OIOs and average read latencies. This ensures that cached writes are not going to affect the overall device model. Second, we ignore data points where an extreme behavior is detected in terms of average IO size and sequentiality. In our current prototype, we ignore data points when IO size is greater than 32 KB or sequentiality is more than 90%. In the future, we plan to study normalizing latency by IO size instead of ignoring such data points. In practice, this isn't a big problem because (a) with virtualization, single LUNs typically host VMs with numerous different workload types, (b) we expect to collect data for each LUN

over a period of days in order to make migration decisions, which allows IO from various VMs to be included in our results and (c) even if a single VM workload is sequential, the overall IO pattern arriving at the array may look random due to high consolidation ratios typical in virtualized systems.

With these provisions in place, we used DVD Store again to perform device modeling and looked at the slope values for two different LUNs with 4 and 8 disks. Figure 9 shows the slope values for the two LUNs. Note that the slopes are positive for both LUNs and the slope is lower for the LUN with more disks.

Cache size available to a LUN can also impact the overall IO performance. The first order impact should be captured by the IO latency seen by a workload. In some experiments, we observed that the slope was smaller for LUNs on an array with a larger cache, even if other characteristics were similar. Next, we complete the algorithm by showing how the workload and device models are used for dynamic load balancing and initial placement of virtual disks on LUNs.

5 Load Balance Engine

Load balancing requires a metric to balance over multiple resources. We use the numerator of Equation 1 (denoted as $\mathscr{L}_i$), as the main metric for load balancing for each workload W_i. Furthermore, we also need to consider LUN performance while doing load balancing. We use parameter $\mathscr{P}_j$ to represent the performance of device D_j. Intuitively we want to make the load proportional to the performance of each device. So the problem reduces to equalizing the ratio of the sum of workload metrics and the LUN performance metric for each LUN. Mathematically, we want to equate the following across devices:

$$\frac{\sum_{\forall\ W_i\ on\ D_j} \mathscr{L}_i}{\mathscr{P}_j} \tag{4}$$

The algorithm first computes the sum of workload metrics. Let N be the normalized load on a device:

$$N_j = \frac{\sum \mathscr{L}_i}{\mathscr{P}_j} \tag{5}$$

Let $Avg(\{N\})$ and $\sigma(\{N\})$ be the average and standard deviation of the normalized load across devices. Let the imbalance fraction f be defined as $f(\{N\}) = \sigma(\{N\})/Avg(\{N\})$. In a loop, until we get the imbalance fraction $f(\{N\})$ under a threshold, we pick the devices with minimum and maximum normalized load to do pairwise migrations such that the imbalance is lowered with each move. Each iteration of the loop tries to find the virtual disks that need to be moved from the device with maximum normalized load to the one with the minimum normalized load. Perfect balancing between these two devices is a variant of subset-sum problem which is known to be NP-complete. We are using one of the approximations [16] proposed for this problem with a quite good competitive ratio of 3/4 with respect to optimal. We have tested other heuristics as well, but the gain from trying to reach the best balance is outweighed by the cost of migrations in some cases.

Algorithm 1: Load Balancing Step

```
foreach device D_j do
    foreach workload W_i currently placed D_j do
        S+ = L_i
    N_j ⟵ S/P_j
while f({N}) > imbalanceThreshold do
    d_x ⟵ Device with maximum normalized load
    d_y ⟵ Device with minimum normalized load
    N_x, N_y ⟵ PairWiseRecommendMigration(d_x, d_y)
```

Algorithm 1 presents the pseudo-code for the load balancing algorithm. The imbalance threshold can be used to control the tolerated degree of imbalance in the system and therefore the aggressiveness of the algorithm. Optimizations in terms of data movement and cost of migrations are explained next.

Workload/Virtual Disk Selection: To refine the recommendations, we propose biasing the choice of migration candidates in one of many ways: (1) pick virtual disks with the highest value of $\mathscr{L}_i/(disk\ size)$ first, so that the change in load per GB of data movement is higher leading to smaller data movement, (2) pick virtual disks with smallest current IOPS/$\mathscr{L}_i$ first, so that the immediate impact of data movement is minimal, (3) filter for constraints such as affinity between virtual disks and data stores, (4) avoid ping-ponging of the same virtual disk between data stores, (5) prevent migration movements that violate per-VM data reliability or data protection policies (*e.g.*, RAID-level), etc. Hard constraints (*e.g.*, access to the destination data store at the current host running the VM) can also be handled as part of virtual disk selection in this step. Overall, this step incorporates any cost-benefit analysis that is needed to choose which VMs to migrate in order to do load balancing. After computing these recommendations, they can either be presented to the user as suggestions or can be carried out automatically during periods of low activity. Administrators can even configure the times when the migrations should be carried out, *e.g.*, migrate on Saturday nights after 2am.

Initial Placement: A good decision for the initial placement of a workload is as important as future migrations. Initial placement gives us a good way to reduce potential imbalance issues in future. In BASIL, we use the over-

all normalized load N as an indicator of current load on a LUN. After resolving user-specified hard constraints (*e.g.*, reliability), we choose the LUN with the minimum value of the normalized load for a new virtual disk. This ensures that with each initial placement, we are attempting to naturally reduce the overall load imbalance among LUNs.

Discussion: In previous work [12], we looked at the impact of consolidation on various kinds of workloads. We observed that when random workloads and the underlying devices are consolidated, they tend to perform at least as good or better in terms of handling bursts and the overall impact of interference is very small. However, when random and sequential workloads were placed together, we saw degradation in throughput of sequential workloads. As noted in Section 3, studies [12, 13] of several enterprise applications such as Microsoft Exchange and databases have observed that random access IO patterns are the predominant type.

Nevertheless, to handle specific workloads such as log virtual disks, decision support systems, and multi-media servers, we plan to incorporate two optimizations. First, identifying such cases and isolating them on a separate set of spindles to reduce interference. Second, allocating fewer disks to the sequential workloads because their performance is less dependent on the number of disks as compared to random ones. This can be done by setting soft affinity for these workloads to specific LUNs, and anti-affinity for them against random ones. Thus we can bias our greedy load balancing heuristic to consider such affinity rules while making placement decisions.

Whereas we consider these optimizations as part of our future work, we believe that the proposed techniques are useful for a wide variety of cases, even in their current form, since in some cases, administrators may isolate such workloads on separate LUNs manually and set hard affinity rules. We can also assist storage administrators by identifying such workloads based on our online data collection. In some cases users may have reliability or other policy constraints such as RAID-level or mirroring, attached to VM disks. In those cases a set of devices would be unsuitable for some VMs, and we would treat that as a hard constraint in our load balancing mechanism while recommending placements and migrations. Essentially the migrations would occur among devices with similar static characteristics. The administrator can choose the set of static characteristics that are used for combining devices into a single storage POD (our load balancing domain). Some of these may be reliabilitity, backup frequency, support for de-duplication, thin provisioning, security isolation and so on.

Type	OIO range	IO size	%Read	%Random
Workstation	[4-12]	8	80	80
Exchange	[4-16]	4	67	100
OLTP	[12-16]	8	70	100
Webserver	[1-4]	4	95	75

Table 1: Iometer workload configuration definitions.

6 Experimental Evaluation

In this section we discuss experimental results based on an extensive evaluation of BASIL in a real testbed. The metrics that we use for evaluating BASIL are overall throughput gain and overall latency reduction. Here overall throughput is aggregated across all data stores and overall latency is the average latency weighted by IOPS across all data stores. These metrics are used instead of just individual data store values, because a change at one data store may lead to an inverse change on another, and our goal is to improve the overall performance and utilization of the system, and not just individual data stores.

6.1 Testing Framework

Since the performance of a storage device depends greatly on the type of workloads to which it is subjected, and their interference, it would be hard to reason about a load balancing scheme with just a few representative test cases. One can always argue that the testing is too limited. Furthermore, once we make a change in the modeling techniques or load balancing algorithm, we will need to validate and compare the performance with the previous versions. To enable repeatable, extensive and quick evaluation of BASIL, we implemented a testing framework emulating a real data center environment, although at a smaller scale. Our framework consists of a set of hosts, each running multiple VMs. All the hosts have access to all the data stores in the load balancing domain. This connectivity requirement is critical to ensure that we don't have to worry about physical constraints during our testing. In practice, connectivity can be treated as another migration constraint. Our testing framework has three modules: admin, modeler and analyzer that we describe in detail next.

Admin module: This module initiates the workloads in each VM, starts collecting periodic IO stats from all hosts and feeds the stats to the next module for generation of workload and device models. The IO stats are collected per virtual disk. The granularity of sampling is configurable and set to 2-10 seconds for experiments in this paper. Finally, this module is also responsible for applying migrations that are recommended by the analyzer. In order to speed up the testing, we emulate the migrations by shifting the workload from one data store to another, instead of actually doing data migration. This is possible because we create an identical copy of each virtual disk

		Before Running BASIL			After Running BASIL		
Iometer Workload	**BASIL Online Workload Model [OIO, IOsize, Read%, Random%]**	**Latency (ms)**	**Throughput (IOPS)**	**Location**	**Latency (ms)**	**Throughput (IOPS)**	**Location**
oltp	[7, 8, 70, 100]	28	618	3diskLUN	22	1048	3diskLUN
oltp	[16, 8, 69, 100]	35	516	3diskLUN	12	1643	9diskLUN
workstation	[6, 8, 81, 79]	60	129	3diskLUN	24	338	9diskLUN
exchange	[6, 4, 67, 100]	9	940	6diskLUN	9	964	6diskLUN
exchange	[6, 4, 67, 100]	11	777	6diskLUN	8	991	6diskLUN
workstation	[4, 8, 80, 79]	13	538	6diskLUN	21	487	9diskLUN
webserver	[1, 4, 95, 74]	4	327	9diskLUN	29	79	9diskLUN
webserver	[1, 4, 95, 75]	4	327	9diskLUN	45	81	9diskLUN
Weighted Average Latency or Total Throughput		16.7	4172		14.9 (-11%)	5631 (+35%)	

Table 2: BASIL online workload model and recommended migrations for a sample initial configuration. Overall average latency and IO throughput improved after migrations.

			Before BASIL		After BASIL	
Data Stores	**# Disks**	$\mathscr{P}$ **=*1/Slope***	**Latency (ms)**	**IOPS**	**Latency (ms)**	**IOPS**
3diskLUN	3	0.7	34	1263	22	1048
6diskLUN	6	1.4	10	2255	8	1955
9diskLUN	9	2.0	4	654	16	2628

Table 3: BASIL online device model and disk migrations for a sample initial configuration. Latency, IOPS and overall load on three data stores before and after recommended migrations.

on all data stores, so a VM can just start accessing the virtual disk on the destination data store instead of the source one. This helped to reduce our experimental cycle from weeks to days.

Modeler: This module gets the raw stats from the admin module and creates both workload and device models. The workload models are generated by using per virtual disk stats. The module computes the cumulative distribution of all four parameters: OIOs, IO size, Read% and Random%. To compute the workload load metric $\mathscr{L}_i$, we use the 90th percentile values of these parameters. We didn't choose average values because storage workloads tend to be bursty and the averages can be much lower and more variable compared to the 90th percentile values. We want the migration decision to be effective in most cases instead of just average case scenarios. Since migrations can take hours to finish, we want the decision to be more conservative rather than aggressive.

For the device models, we aggregate IO stats from different hosts that may be accessing the same device (*e.g.*, using a cluster file system). This is very common in virtualized environments. The OIO values are aggregated as a sum, and the latency value is computed as a weighted average using IOPS as the weight in that interval. The $\langle OIO, Latency \rangle$ pairs are collected over a long period of time to get higher accuracy. Based on these values, the modeler computes a slope $\mathscr{P}_i$ for each device. A device with no data, is assigned a slope of zero which also mimics the introduction of a new device in the POD.

Analyzer: This module takes all the workload and device models as input and generates migration recommendations. It can also be invoked to perform initial placement of a new virtual disk based on the current configuration. The output of the analyzer is fed into the admin module to carry out the recommendations. This can be done iteratively till the load imbalance is corrected and the system stabilizes with no more recommendations generated.

The experiments presented in the next sections are run on two different servers, one configured with 2 dual-core 3 GHz CPUs, 8 GB RAM and the other with 4 dual-core 3 GHz CPUs and 32 GB RAM. Both hosts have access to three data stores with 3, 6 and 9 disks over a FC SAN network. These data stores are 150 GB in size and are created on an EMC CLARiiON storage array. We ran 8 VMs for our experiments each with one 15 GB OS disk and one 10 GB experimental disk. The workloads in the VMs are generated using Iometer [3]. The Iometer workload types are selected from Table 1, which shows Iometer configurations that closely represent some of the real enterprise workloads [5].

6.2 Simple Load Balancing Scenario

In this section, we present detailed analysis for one of the input cases which looks balanced in terms of number of VMs per data store. Later, we'll also show data for a large number of other scenarios. As shown in Table 2, we started with an initial configuration using 8 VMs, each running a workload chosen from Table 1 against one of the three data stores. First we ran the workloads in VMs without BASIL; Table 2 shows the corresponding throughput (IOPS) and latency values seen by the workloads. Then we ran BASIL, which created workload and device models online. The computed workload model is shown in the second column of Table 2 and device model is shown as $\mathscr{P}$ (third column) in Table 3. It is worth noting that the computed performance metrics for

		Before Running BASIL			After Running BASIL		
Iometer Workload	**BASIL Online Workload Model [OIO, IOsize, Read%, Random%]**	**Latency (ms)**	**Throughput (IOPS)**	**Location**	**Latency (ms)**	**Throughput (IOPS)**	**Location**
exchange	[8, 4, 67, 100]	37	234	6diskLUN	62	156	6diskLUN
exchange	[8, 4, 67, 100]	39	227	6diskLUN	12	710	3diskLUN
webserver	[2, 4, 95, 75]	54	43	6diskLUN	15	158	9diskLUN
webserver	[2, 4, 95, 75]	60	39	6diskLUN	18	133	9diskLUN
workstation	[7, 8, 80, 80]	41	191	6diskLUN	11	657	9diskLUN
workstation	[8, 8, 80, 80]	51	150	6diskLUN	11	686	9diskLUN
oltp	[8, 8, 70, 100]	64	402	6diskLUN	28	661	6diskLUN
oltp	[8, 8, 70, 100]	59	410	6diskLUN	28	658	6diskLUN
Weighted Average Latency or Total Throughput		51.6	1696		19.5 (-62%)	3819 (+125%)	

Table 4: New device provisioning: 3DiskLUN and 9DiskLUN are newly added into the system that had 8 workloads running on the 6DiskLUN. Average latency, IO throughput and placement for all 8 workloads before and after migration.

			Before BASIL		After BASIL	
Data Stores	**# Disks**	$\mathscr{P}$ *=1/Slope*	**Latency (ms)**	**IOPS**	**Latency (ms)**	**IOPS**
3diskLUN	3	0.6	0	0	12	710
6diskLUN	6	1.4	51	1696	31	1475
9diskLUN	9	1.7	0	0	11	1634

Table 5: New device provisioning: latency, IOPS and overall load on three data stores.

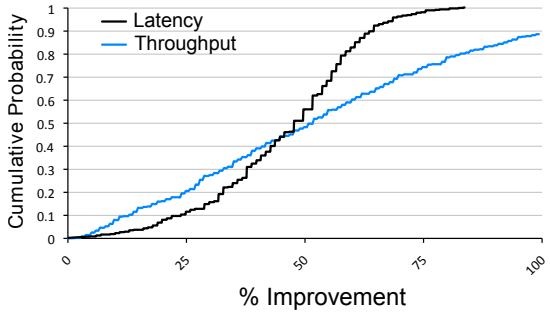


Figure 10: CDF of throughput and latency improvements with load balancing, starting from random configurations.

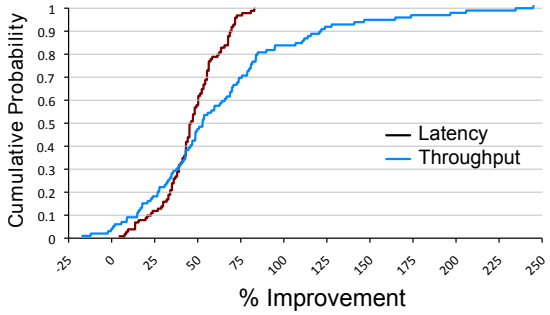


Figure 11: CDF of latency and throughput improvements from BASIL initial placement versus random.

devices are proportional to their number of disks. Based on the modeling, BASIL suggested three migrations over two rounds. After performing the set of migrations we again ran BASIL and no further recommendations were suggested. Tables 2 and 3 show the performance of workloads and data stores in the final configuration. Note that 5 out of 8 workloads observed an improvement in IOPS and reduction in latency. The aggregated IOPS across all data stores (shown in Table 2) improved by 35% and overall weighted latency decreased by 11%. This shows that for this sample setup BASIL is able to recommend migrations based on actual workload characteristics and device modeling, thereby improving the overall utilization and performance.

6.3 New Device Provisioning

Next we studied the behavior of BASIL during the well known operation of adding more storage devices to a storage POD. This is typically in response to a space crunch or a performance bottleneck. In this experiment, we started with all VMs on the single 6DiskLUN data store and we added the other two LUNs into the system. In the first round, BASIL observed the two new data stores, but didn't have any device model for them due to lack of IOs. In a full implementation, we have the option of performing some offline modeling at the time of provisioning, but currently we use the heuristic of placing only one workload on a new data store with no model.

Table 4 shows the eight workloads, their computed models, initial placement and the observed IOPS and latency values. BASIL recommended five migrations over two rounds. In the first round BASIL migrated one workload to each of 3DiskLUN and 9DiskLUN. In the next round, BASIL had slope information for all three data stores and it migrated three more workloads from 6DiskLUN to 9DiskLUN. The final placement along with performance results are again shown in Table 4. Seven out of eight workloads observed gains in throughput and decreased latencies. The loss in one workload is offset by gains in others on the same data store. We believe

that this loss happened due to unfair IO scheduling of LUN resources at the storage array. Such effects have been observed before [11]. Overall data store models and performance before and after running BASIL are shown in Table 5. Note that the load is evenly distributed across data stores in proportion to their performance. In the end, we observed a 125% gain in aggregated IOPS and 62% decrease in weighted average latency (Table 4). This shows that BASIL can handle provisioning of new storage devices well by quickly performing online modeling and recommending appropriate migrations to get higher utilization and better performance from the system.

6.4 Summary for 500 Configurations

Having looked at BASIL for individual test cases, we ran it for a large set of randomly generated initial configurations. In this section, we present a summary of results of over 500 different configurations. Each test case involved a random selection of 8 workloads from the set shown in Table 1, and a random initial placement of them on three data stores. Then in a loop we collected all the statistics in terms of IOPS and latency, performed online modeling, ran the load balancer and performed workload migrations. This was repeated until no further migrations were recommended. We observed that all configurations showed an increase in overall IOPS and decrease in overall latency. There were fluctuations in the performance of individual workloads, but that is expected given that load balancing puts extra load on some data stores and reduces load on others. Figure 10 shows the cumulative distribution of gain in IOPS and reduction in latency for 500 different runs. We observed an overall throughput increase of greater than 25% and latency reduction of 33% in over 80% of all the configurations that we ran. In fact, approximately half the tests cases saw at least 50% higher throughput and 50% better latency. This is very promising as it shows that BASIL can work well for a wide range of workload combinations and their placements.

6.5 Initial Placement

One of the main use cases of BASIL is to recommend initial placement for new virtual disks. Good initial placement can greatly reduce the number of future migrations and provide better performance from the start. We evaluated our initial placement mechanism using two sets of tests. In the first set we started with one virtual disk, placed randomly. Then in each iteration we added one more disk into the system. To place the new disk, we used the current performance statistics and recommendations generated by BASIL. No migrations were computed by BASIL; it ran only to suggest initial placement.

Workload	BASIL Online Workload Model [OIO, IOsize, Read%, Random%]
dvdstore-1	[5, 8, 100, 100]
dvdstore-2	[3, 62, 100, 100]
dvdstore-3	[6, 8, 86, 100]
swing-1	[13, 16, 67, 100]
swing-2	[31, 121, 65, 100]
fb-mail-1	[4, 5, 16, 99]
fb-mail-2	[5, 6, 52, 99]
fb-mail-3	[7, 6, 47, 99]
fb-mail-4	[5, 5, 60, 99]
fb-oltp-1	[1, 2, 100, 100]
fb-oltp-2	[6, 8, 86, 100]
fb-web-1	[8, 18, 99, 98]
fb-web-2	[5, 5, 60, 99]

Table 6: Enterprise workloads. For the database VMs, only the table space and index disks were modeled.

Data Stores	# Disks	RAID	LUN Size	$\mathscr{P}$ =*1/Slope*
EMC	6 FC	5	450 GB	1.1
NetApp-SP	7 FC	5	400 GB	0.83
NetApp-DP	7 SATA	6	250 GB	0.48

Table 7: Enterprise workload LUNs and their models.

We compared the performance of placement done by BASIL with a random placement of virtual disks as long as space constraints were satisfied. In both cases, the VMs were running the exact same workloads. We ran 100 such cases, and Figure 11 shows the cumulative distribution of percentage gain in overall throughput and reduction in overall latency of BASIL as compared to random selection. This shows that the placement recommended by BASIL provided 45% reduction in latency and 53% increase in IOPS for at least half of the cases, as compared to the random placement.

The second set of tests compare BASIL with an oracle that can predict the best placement for the next virtual disk. To test this, we started with an initial configuration of 7 virtual disks that were randomly chosen and placed. We ran this configuration and fed the data to BASIL to find a data store for the eighth disk. We tried the eighth disk on all the data stores manually and compared the performance of BASIL's recommendation with the best possible placement. To compute the rank of BASIL compared to the oracle, we ran 194 such cases and BASIL chose the best data store in 68% of them. This indicates that BASIL finds good initial placements with high accuracy for a wide variety of workload configurations.

6.6 Enterprise Workloads

In addition to the extensive micro-benchmark evaluation, we also ran enterprise applications and filebench workload models to evaluate BASIL in more realistic scenarios. The CPU was not bottlenecked in any of the experiments. For the database workloads, we isolated the data and log virtual disks. Virtual disks containing data

Workload	T	Space-Balanced			After Two BASIL Rounds			Human Expert #1			Human Expert #2		
	Units	R	T	Location	R	T	Location	R	T	Location	R	T	Location
dvd-1	opm	72	2753	EMC	78	2654	EMC	59	2986	EMC	68	2826	NetApp-SP
dvd-2	opm	82	1535	NetApp-SP	89	1487	*EMC*	58	1706	EMC	96	1446	EMC
dvd-3	opm	154	1692	NetApp-DP	68	2237	*NetApp-SP*	128	1821	NetApp-DP	78	2140	EMC
swing-1	tpm	n/r	8150	NetApp-SP	n/r	8250	NetApp-SP	n/r	7500	NetApp-DP	n/r	7480	NetApp-SP
swing-2	tpm	n/r	8650	EMC	n/r	8870	EMC	n/r	8950	EMC	n/r	8500	NetApp-DP
fb-mail-1	ops/s	38	60	NetApp-SP	36	63	NetApp-SP	35	61	NetApp-SP	15	63	EMC
fb-mail-2	ops/s	35	84	NetApp-SP	37	88	NetApp-SP	34	85	NetApp-SP	16	88	EMC
fb-mail-3	ops/s	81	67	NetApp-DP	27	69	NetApp-DP	30	73	NetApp-SP	28	74	NetApp-SP
fb-mail-4	ops/s	9.2	77	EMC	14	75	EMC	11	76	EMC	16	75	EMC
fb-oltp-1	ops/s	32	25	NetApp-SP	35	25	NetApp-SP	70	24	NetApp-DP	44	25	NetApp-DP
fb-oltp-2	ops/s	84	22	NetApp-DP	40	22	NetApp-DP	79	22	NetApp-DP	30	23	NetApp-SP
fb-web-1	ops/s	58	454	NetApp-DP	26	462	*NetApp-SP*	56	460	NetApp-DP	22	597	EMC
fb-web-2	ops/s	11	550	EMC	11	550	EMC	21	500	NetApp-SP	14	534	EMC

Table 8: Enterprise Workloads. Human expert generated placements versus BASIL. Applying BASIL recommendations resulted in improved application as well as more balanced latencies. *R* denotes application-reported transaction response time (ms) and *T* is the throughput in specified units.

	Space-Balanced		After Two BASIL Rounds		Human Expert #1		Human Expert #2	
	Latency (ms)	IOPS	Latency (ms)	IOPS	Latency (ms)	IOPS	Latency (ms)	IOPS
EMC	9.6	836	12	988	9.9	872	14	781
NetApp-SP	29	551	19	790	27	728	26	588
NetApp-DP	45	412	23	101	40	317	17	340
Weighted Average Latency or Total Throughput	23.6	1799	15.5	1874	21.2	1917	18.9	1709

Table 9: Enterprise Workloads. Aggregate statistics on three LUNs for BASIL and human expert placements.

were placed on the LUNs under test and log disks were placed on a separate LUN. We used five workload types as explained below.

DVDStore [1] version 2.0 is an online e-commerce test application with a SQL database, and a client load generator. We used a 20 GB dataset size for this benchmark, 10 user threads and 150 ms think time between transactions.

Swingbench [4] (order entry workload) represents an online transaction processing application designed to stress an underlying Oracle database. It takes the number of users, think time between transactions, and a set of transactions as input to generate a workload. For this workload, we used 50 users, 100-200 ms think time between requests and all five transaction types (*i.e.*, new customer registration, browse products, order products, process orders and browse orders with variable percentages set to 10%, 28%, 28%, 6% and 28% respectively).

Filebench [2], a well-known application IO modeling tool, was used to generate three different types of workloads: OLTP, mail server and webserver.

We built 13 VMs running different configurations of the above workloads as shown in Table 6 and ran them on two quad-core servers with 3 GHz CPUs and 16 GB RAM. Both hosts had access to three LUNs with different characteristics, as shown in Table 7. To evaluate BASIL's performance, we requested domain experts within VMware to pick their own placements using full knowledge of workload characteristics and detailed knowledge of the underlying storage arrays. We requested two types of configurations: space-balanced and performance-balanced.

The space-balanced configuration was used as a baseline and we ran BASIL on top of that. BASIL recommended three moves over two rounds. Table 8 provides the results in terms of the application-reported transaction latency and throughput in both configurations. In this instance, the naive space-balanced configuration had placed similar load on the less capable data stores as on the faster ones causing VMs on the former to suffer from higher latencies. BASIL recommended moves from less capable LUNs to more capable ones, thus balancing out application-visible latencies. This is a key component of our algorithm. For example, before the moves, the three DVDStore VMs were seeing latencies of 72 ms, 82 ms and 154 ms whereas a more balanced result was seen afterward: 78 ms, 89 ms and 68 ms. Filebench OLTP workloads had a distribution of 32 ms and 84 ms before versus 35 ms and 40 ms afterward. Swingbench didn't report latency data but judging from the throughput, both VMs were well balanced before and BASIL didn't change that. The Filebench webserver and mail VMs also had much reduced variance in latencies. Even compared to the two expert placement results, BASIL fares better in terms of variance. This demonstrates the ability of BASIL to balance real enterprise workloads across data stores of very different capabilities using online models.

BASIL also performed well in the critical metrics of maintaining overall storage array efficiency while balanc-

ing load. Table 9 shows the achieved device IO latency and IO throughput for the LUNs. Notice that, in comparison to the space-balanced placement, the weighted average latency across three LUNs went down from 23.6 ms to 15.5 ms, a gain of 34%, while IOPS increased slightly by 4% from 1799 to 1874. BASIL fared well even against hand placement by domain experts. Against expert #2, BASIL achieved an impressive 18% better latency and 10% better throughput. Compared to expert #1, BASIL achieved a better weighted average latency by 27% albeit with 2% less throughput. Since latency is of primary importance to enterprise workloads, we believe this is a reasonable trade off.

7 Conclusions and Future Work

This paper presented BASIL, a storage management system that does initial placement and IO load balancing of workloads across a set of storage devices. BASIL is novel in two key ways: (1) identifying IO latency as the primary metric for modeling, and (2) using simple models both for workloads and devices that can be efficiently obtained online. The linear relationship of IO latency with various parameters such as outstanding IOs, IO size, read % etc. is used to create models. Based on these models, the load balancing engine recommends migrations in order to balance load on devices in proportion to their capabilities.

Our extensive evaluation in a real system with multiple LUNs and workloads shows that BASIL achieved improvements of at least 25% in throughput and 33% in overall latency in over 80% of the hundreds of micro-benchmark configurations that we tested. Furthermore, for real enterprise applications, BASIL lowered the variance of latencies across the workloads and improved the weighted average latency by 18-27% with similar or better achieved throughput when evaluated against configurations generated by human experts.

So far we've focused on the quality of the BASIL recommended moves. As future work, we plan to add migration cost considerations into the algorithm and more closely study convergence properties. Also on our roadmap is special handling of the less common sequential workloads, as well as applying standard techniques for ping-pong avoidance. We are also looking at using automatically-generated affinity and anti-affinity rules to minimize the interference among various workloads accessing a device.

Acknowledgments

We would like to thank our shepherd Kaladhar Voruganti for his support and valuable feedback. We are grateful to Carl Waldspurger, Minwen Ji, Ganesha Shanmuganathan, Anne Holler and Neeraj Goyal for valuable discussions and feedback. Thanks also to Keerti Garg, Roopali Sharma, Mateen Ahmad, Jinpyo Kim, Sunil Satnur and members of the performance and resource management teams at VMware for their support.

References

[1] DVD Store. http://www.delltechcenter.com/page/DVD+store.

[2] Filebench. http://solarisinternals.com/si/tools/filebench/index.php.

[3] Iometer. http://www.iometer.org.

[4] Swingbench. http://www.dominicgiles.com/swingbench.html.

[5] Workload configurations for typical enterprise workloads. http://blogs.msdn.com/tvoellm/archive/2009/05/07/useful-io-profiles-for-simulating-various-workloads.aspx.

[6] Resource Management with VMware DRS, 2006. http://vmware.com/pdf/vmware_drs_wp.pdf.

[7] AHMAD, I. Easy and Efficient Disk I/O Workload Characterization in VMware ESX Server. *IISWC* (Sept. 2007).

[8] ALVAREZ, G. A., AND ET AL. Minerva: an automated resource provisioning tool for large-scale storage systems. In *ACM Transactions on Computer Systems* (Nov. 2001).

[9] ANDERSON, E. Simple table-based modeling of storage devices. Tech. rep., SSP Technical Report, HP Labs, July 2001.

[10] ANDERSON, E., AND ET AL. Hippodrome: running circles around storage administration. In *Proc. of Conf. on File and Storage Technology (FAST'02)* (Jan. 2002).

[11] GULATI, A., AHMAD, I., AND WALDSPURGER, C. PARDA: Proportionate Allocation of Resources for Distributed Storage Access. In *USENIX FAST* (Feb. 2009).

[12] GULATI, A., KUMAR, C., AND AHMAD, I. Storage Workload Characterization and Consolidation in Virtualized Environments. In *Workshop on Virtualization Performance: Analysis, Characterization, and Tools (VPACT)* (2009).

[13] KAVALANEKAR, S., WORTHINGTON, B., ZHANG, Q., AND SHARDA, V. Characterization of storage workload traces from production windows servers. In *IEEE IISWC* (Sept. 2008).

[14] MERCHANT, A., AND YU, P. S. Analytic modeling of clustered raid with mapping based on nearly random permutation. *IEEE Trans. Comput. 45*, 3 (1996).

[15] MESNIER, M. P., WACHS, M., SAMBASIVAN, R. R., ZHENG, A. X., AND GANGER, G. R. Modeling the relative fitness of storage. *SIGMETRICS Perform. Eval. Rev. 35*, 1 (2007).

[16] PRZYDATEK, B. A Fast Approximation Algorithm for the Subset-Sum Problem, 1999.

[17] RUEMMLER, C., AND WILKES, J. An introduction to disk drive modeling. *IEEE Computer 27*, 3 (1994).

[18] SHEN, Y.-L., AND XU, L. An efficient disk I/O characteristics collection method based on virtual machine technology. *10th IEEE Intl. Conf. on High Perf. Computing and Comm.* (2008).

[19] SHRIVER, E., MERCHANT, A., AND WILKES, J. An analytic behavior model for disk drives with readahead caches and request reordering. *SIGMETRICS Perform. Eval. Rev. 26*, 1 (1998).

[20] UYSAL, M., ALVAREZ, G. A., AND MERCHANT, A. A modular, analytical throughput model for modern disk arrays. In *MASCOTS* (2001).

[21] VARKI, E., MERCHANT, A., XU, J., AND QIU, X. Issues and challenges in the performance analysis of real disk arrays. *IEEE Trans. Parallel Distrib. Syst. 15*, 6 (2004).

[22] WANG, M., AU, K., AILAMAKI, A., BROCKWELL, A., FALOUTSOS, C., AND GANGER, G. R. Storage Device Performance Prediction with CART Models. In *MASCOTS* (2004).

Discovery of Application Workloads from Network File Traces

Neeraja J. Yadwadkar, Chiranjib Bhattacharyya, K. Gopinath
Department of Computer Science and Automation, Indian Institute of Science

Thirumale Niranjan, Sai Susarla
NetApp Advanced Technology Group

Abstract

An understanding of application I/O access patterns is useful in several situations. First, gaining insight into what applications are doing with their data at a semantic level helps in designing efficient storage systems. Second, it helps create benchmarks that mimic realistic application behavior closely. Third, it enables autonomic systems as the information obtained can be used to adapt the system in a closed loop.

All these use cases require the ability to extract the application-level semantics of I/O operations. Methods such as modifying application code to associate I/O operations with semantic tags are intrusive. It is well known that network file system traces are an important source of information that can be obtained non-intrusively and analyzed either online or offline. These traces are a sequence of primitive file system operations and their parameters. Simple counting, statistical analysis or deterministic search techniques are inadequate for discovering application-level semantics in the general case, because of the inherent variation and noise in realistic traces.

In this paper, we describe a trace analysis methodology based on Profile Hidden Markov Models*. We show that the methodology has powerful discriminatory capabilities that enable it to recognize applications based on the patterns in the traces, and to mark out regions in a long trace that encapsulate sets of primitive operations that represent higher-level application actions. It is robust enough that it can work around discrepancies between training and target traces such as in length and interleaving with other operations. We demonstrate the feasibility of recognizing patterns based on a small sampling of the trace, enabling faster trace analysis. Preliminary experiments show that the method is capable of learning accurate profile models on live traces in an online setting. We present a detailed evaluation of this methodology in a UNIX environment using NFS traces of selected commonly used applications such as compilations as well as on industrial strength benchmarks such as TPC-C and Postmark, and discuss its capabilities and limitations in the context of the use cases mentioned above.*

1 Introduction

Enterprise systems require an understanding of the behavior of the applications that use their services. This application-level knowledge is necessary for self-tuning, planning or automated troubleshooting and management. Unfortunately, there is no accepted mechanism for this knowledge to flow from the application to the system. We can neither impose upon application developers to give hints, nor over-engineer network protocols to transport more semantics. Therefore, we need mechanisms for systems to *learn* what the application is doing *automatically*.

Being able to identify the application-level workload has significant benefits. If we can figure out that the client OLTP (online transaction processing) application is doing a *join*, we can tune the caching and prefetching suitably. If we can discover that the client is executing the *compile* phase of a *make*, we can immediately know that it will be followed by a *link* phase, that the output files generated will be accessed very soon, and that the output files can be placed on less-critical storage since they can be generated at will. If we can spot that the client is executing a *copy* operation, then we can derive data provenance information usable by compliance engines. If we can match the signature of a trace with that of known malware or viruses, that can be useful as well. We can employ offline workload identification for auditing, forensics and chargeback. We can help storage systems management by providing inputs to sizing and planning tools.

In this paper, we tackle a specific instance of the problem – given the headers of an NFS [4] trace, to identify the application-level workload that generated it. NFS clients send messages to the server that contain opcodes such as READ, WRITE, SETATTR, READDIR, etc., their associated parameters such as file handles and file offsets, and data. An NFS trace contains a timestamped sequence of these messages along with the responses sent by the server to the client. These traces can be easily captured [12, 1] for online or offline analysis, allowing us to develop a non-invasive tool using the methodology described here. Furthermore, the NFS trace contains all the interactions between the clients and the server. As all the necessary in-

formation is available, we can assert that any deficiency in tackling our use cases is solely due to the sophistication of the analysis methods.

However, given a trace captured at the server, it is non-trivial to identify the client applications that generated it. First, there could be noise in the form of background communication between the client and server. Second, messages could be interleaved with those from other applications on the same client machine. Third, the application's parameters may create variations in the trace. For instance, traces of a single file copy and that of a recursive file copy may look very different (see Tables 1 and 2), even though it is the same application. Fourth, the asynchrony in multi-threaded applications impact the ordering of messages in the traces. Therefore, we believe that deterministic pattern searching methods will not be able to unearth the fundamental patterns hidden in a trace. Methods originating in the Machine Learning domain have shown considerable promise in computational biology [16, 14] as well as in initial studies on trace analysis [19]. In this paper, we apply a well-known technique called *Profile Hidden Markov Model* (profile HMM) [16, 14] to this problem, and demonstrate its pattern-recognition capabilities with respect to our use cases.

The key contributions of this paper are as follows:

Workload Identification We show that profile HMMs, once trained, are capable of identifying the application that generated the trace. Using commonly used UNIX commands such as *make, cp, find, mv, tar, untar*, etc., as well as industry benchmarks such as TPC-C, we show that we are able to cleanly distinguish the traces that these commands generate.

Trace Annotation We show that our methodology is able to identify transitions between workloads, and mark workload-specific regions in a long trace sequence.

Trace Sampling We show that profile HMMs do not need the entire trace to work on. With merely a 20% segment of the trace, sampled randomly, we are able to discriminate between many workloads and identify them with high confidence. This will enable us to perform faster analysis. Further, we show how to use this ability to identify concurrently executing workloads.

Automated Learning We demonstrate a technique by which the profile HMMs can be trained automatically without manual labeling of workloads. We use the technique to train and then subsequently identify constituent workloads of a Linux kernel compilation task.

Power of Opcode Sequences We show that opcode sequences alone contain sufficient information to tackle many of the common use cases. Other information in the traces such as file handles and offsets are not sufficiently amenable to mathematical modeling, so this result is valuable.

Since the technique we use requires training on data sets followed by a recognition phase and also involves reasonable amounts of computation, it is best suited for those problems whose natural time constants are in the minutes or hours range (such as in system management, for example, detecting configuration errors). Algorithmic approaches, widely used, are still the best if the time constants are much smaller (such as in milliseconds or seconds).

The rest of the paper is organized as follows. Section 2 presents the current state of research in this area and places our work in context. Section 3 describes the mathematics behind our methodology, the workflow associated with it, and describes how it is used to identify workloads and mark out regions exhibiting known patterns in the trace. Section 4 offers experimental validation of our techniques. Finally, Section 6 summarizes our conclusions and proposes avenues for continuing this work.

2 Related Work

There is a rich body of work in which file system traces have been analyzed to get aggregate information about systems and to understand how storage is used over time [2, 17, 24, 11]. Our work differs from this body of work in that we focus on individual workloads running on the system and attempt to discover them. Since prior research efforts are oriented towards extracting gross behavior, counting-based tools suffice. The problem that we tackle in this paper requires more powerful methods.

Traces are a good source of information as they contain a complete picture of the inputs to a system and at the same time are easy to capture in a non-invasive manner. Ellard [10] makes a strong case that the information in NFS traces can be used to enable system optimizations. HMMs generated from block traces have been used for adaptive prefetching [27]. Traces have been used for file classification [19]. In that work, the authors build a decision tree based system that uses NFS traces to infer correlations between the create-time properties of files such as their names and the dynamic properties such as access patterns and size. In this paper, we do not attempt to classify files and data but focus more on the applications that access them.

The power of HMM as a tool to extract workload access patterns is known [18]. Our work is significantly larger in scope. While they restrict themselves to inferring the sequentiality of workloads using read and write headers in the block traces, we use all the opcodes available in NFS headers to discover the higher-level application that caused it. The sequentiality of a workload can perhaps also be discovered using our framework by including the file offsets as

part of the alphabet through an appropriate scheme of quantization.

Magpie [3] diagnoses problems in distributed systems by monitoring the communications between black-box components, and applying an edit-distance based clustering method to group similar workloads together. Somewhat similar is Spectroscope [25], which uses clustering on request flow graphs constructed from traces to categorize and learn about differences in system behavior. Intrusion detection is another area where various such techniques are used. Warrender [29] surveys methods for intrusion detection using various data mining techniques including HMMs, on system call traces.

Our work is different from all of the above in that it is not only able to identify a higher-level workload, given a trace, but also to be able to accurately mark out workload regions in a composite trace.

3 Methodology

A key observation that motivates our approach to solving the problem is that NFS traces corresponding to a given workload class exhibit significant variability, yet have a characteristic signature. For instance, look at the four traces depicting a *cp* command, shown in Tables 1 and 2. The fuzziness in the repeating subsequences in the trace of *cp * dir/* and *cp -r dir1 dir* make us look at probabilistic methods.

An HMM is appropriate for probabilistic modeling of sequences, and has been used in similar settings in the past [14]. However, in our case, the sequences of the same workload show additions, deletions and mutations between them that are not easily modeled by an HMM. A *cp foo bar* differs from *cp foo dir/* – the latter has an extra *lookup* operation, as seen in Table 2. Our method should have the power to ignore this extra operation since that operation must not be used for discrimination. A variant of the HMM called the profile HMM [8] offers exactly this ability, via *non-emitting* (or *delete*) states. Therefore, we conjecture that profile HMM will be a good method to use for classifying NFS traces. In the rest of this section, we first outline the theory behind the profile HMM and then describe the workflow of our workload identification methodology.

cp * dir/
GETATTR Call, FH:0x0eb18814
READDIRPLUS Call, FH:0x0eb18814
READDIRPLUS Reply (Call In 9) ...
LOOKUP Call, DH:0xe003db8b/tqslwiz.h
LOOKUP Reply Error:NFS3ERR_NOENT
GETATTR Call, FH:0x21b1a714
ACCESS Call, FH:0x21b1a714
CREATE Call, DH:0xe003db8b/tqslwiz.h
SETATTR Call, FH:0x6bd9e67c
GETACL Call
GETATTR Call, FH:0x6bd9e67c
READ Call, FH:0x21b1a714 ...
WRITE Call, FH:0x6bd9e67c ...
COMMIT Call, FH:0x6bd9e67c
GETATTR Call, FH:0xe003db8b
LOOKUP Call, DH:0xe003db8b/TrustedQSL.spec
LOOKUP Reply Error:NFS3ERR_NOENT
GETATTR Call, FH:0x2fb1a914
ACCESS Call, FH:0x2fb1a914
CREATE Call, DH:0xe003db8b/TrustedQSL.spec
SETATTR Call, FH:0x65d9e87c
GETATTR Call, FH:0x65d9e87c
READ Call, FH:0x2fb1a914 ...
WRITE Call, FH:0x65d9e87c ...
COMMIT Call, FH:0x65d9e87c
LOOKUP Call,
DH:0xe003db8b/TrustedQSL.spec.in
LOOKUP Reply Error:NFS3ERR_NOENT
GETATTR Call, FH:0x23b1a514
ACCESS Call, FH:0x23b1a514
CREATE Call,
DH:0xe003db8b/TrustedQSL.spec.in
SETATTR Call, FH:0x67d9ea7c
GETATTR Call, FH:0x67d9ea7c
READ Call, FH:0x23b1a514 ...
WRITE Call, FH:0x67d9ea7c ...
COMMIT Call, FH:0x67d9ea7c

cp -r dir1 dir
ACCESS Call, FH:0xc5914d40
LOOKUP Call, DH:0xc5914d40/dir
LOOKUP Reply Error:NFS3ERR_NOENT
MKDIR Call, DH:0xc5914d40/dir
GETATTR Call, FH:0xc5914d40
GETACL Call
ACCESS Call, FH:0xc5914d40
LOOKUP Call, DH:0xc5914d40/dir
LOOKUP Reply, FH:0x3fb1b914
GETATTR Call, FH:0x0eb18814
ACCESS Call, FH:0x0eb18814
READDIRPLUS Call, FH:0x0eb18814
READDIRPLUS Reply . ..
ACCESS Call, FH:0x3fb1b914
MKDIR Call, DH:0x3fb1b914/hh
GETATTR Call, FH:0x3fb1b914
GETACL Call
GETATTR Call, FH:0x3fb1b914
GETATTR Call, FH:0x36b1b014
ACCESS Call, FH:0x36b1b014
READDIRPLUS Call, FH:0x36b1b014
READDIRPLUS Reply . ..
GETATTR Call, FH:0x39b1bf14
ACCESS Call, FH:0x39b1bf14
ACCESS Call, FH:0x3db1bb14
CREATE Call, DH:0x3db1bb14/contacts.csv
SETATTR Call, FH:0x33b1b514
GETACL Call
GETATTR Call, FH:0x33b1b514
READ Call, FH:0x39b1bf14 ...
WRITE Call, FH:0x33b1b514 ...
COMMIT Call, FH:0x33b1b514
GETATTR Call, FH:0x21b1a714
ACCESS Call, FH:0x21b1a714
CREATE Call, DH:0x3fb1b914/tqslwiz.h
SETATTR Call, FH:0x35b1b314
GETATTR Call, FH:0x35b1b314
READ Call, FH:0x21b1a714 ...
WRITE Call, FH:0x35b1b314 ...
COMMIT Call, FH:0x35b1b314

Table 1. Two *cp* NFS trace headers. The first one copies 3 files into a directory, while the second one is a recursive copy. These traces illustrate that workloads repeat some elements of the trace, with one region being underlined. However, the repetition of symbols is not strict and cannot be captured by a finite state automata model. There is sufficient variability that warrants a fuzzy or probabilistic pattern recognition algorithm such as an HMM. Figure shows only the client→server requests, not the responses. The sole exception is that of responses to LOOKUP since they will help the reader understand the traces.

cp contacts.csv con.csv
ACCESS Call, FH:0xe003db8b
LOOKUP Call, DH:0xe003db8b/con.csv
LOOKUP Reply Error:NFS3ERR_NOENT
LOOKUP Call,
DH:0xe003db8b/contacts.csv
LOOKUP Reply, FH:0x71d9fc7c
GETATTR Call, FH:0x71d9fc7c
ACCESS Call, FH:0x71d9fc7c
CREATE Call, DH:0xe003db8b/con.csv
SETATTR Call, FH:0x58d9d57c
GETACL Call
GETATTR Call, FH:0x58d9d57c
READ Call, FH:0x71d9fc7c ...
WRITE Call, FH:0x58d9d57c ...
COMMIT Call, FH:0x58d9d57c

cp contacts.csv dir/con.csv
LOOKUP Call, DH:0xe003db8b/dir
LOOKUP Reply, FH:0x0eb18814
ACCESS Call, FH:0x0eb18814
LOOKUP Call, DH:0x0eb18814/con.csv
LOOKUP Reply Error:NFS3ERR_NOENT
LOOKUP Call,
DH:0xe003db8b/contacts.csv
LOOKUP Reply, FH:0x71d9fc7c
GETATTR Call, FH:0x71d9fc7c
ACCESS Call, FH:0x71d9fc7c
CREATE Call, DH:0x0eb18814/con.csv
SETATTR Call, FH:0x14b19214
GETACL Call
GETATTR Call, FH:0x14b19214
READ Call, FH:0x71d9fc7c ...
WRITE Call, FH:0x14b19214 ...
COMMIT Call, FH:0x14b19214

Table 2. Two *cp* NFS trace headers. The second one differs from the first in an extra LOOKUP operation (underlined), showing the need for a methodology that can suppress or ignore certain elements in traces. Profile HMM is one such candidate. Figure shows only the client→server requests, not the responses. The sole exception is that of responses to LOOKUP since they will help the reader understand the traces.

3.1 Profile HMMs for Modeling Opcode Traces

It is well known and empirically verified, e.g., Table 1, that opcode traces of the same command are often very similar but not exactly the same. It is also known that traces corresponding to different commands are dissimilar. These observations motivate the development of mathematical models that are capable of discovering a command/workload by merely looking at the trace it generates (e.g., opcode se-

quence), and checking for its similarity with prior traces of the same command with various arguments. The problem of constructing such models is complicated as there is no unique trace for every command. Similar issues arise in many other areas, notable among them being computational biology. The study of designing efficient sequence matching algorithms has received a significant impetus from computational biology where one needs to align a family of many closely related sequences (typically genetic or protein sequences). These sequences diverge due to chance mutations at certain points in the sequence while, at the same time, conserving critical parts of the sequence.

The similarity of two symbol sequences can be measured via the number of mutations needed to make them identical, also called the *edit distance*. Hence, to measure the similarity of a sequence to a set of sequences, one could first align them to be of the same length by adding, deleting or replacing the minimal number of symbols, and then use the smallest edit distance.

As of today there are quite a few techniques for sequence matching, ranging from deterministic [13] to probabilistic approaches [6]. Deterministic approaches are based on dynamic programming, which often leads to algorithms that have prohibitively high time complexity for large symbol sequences: $O(N^r)$ to match with r sequences, each of length N. Probabilistic approaches such as Profile HMMs [6] have emerged as faster alternatives to deterministic methods and have been proven to be very effective for computational biology problems. The key observation behind our work is that trace-based workload identification and annotation maps well to the sequence-matching problem in computational biology, and hence can benefit from similar techniques. Profile HMMs are special Hidden Markov models (HMMs) developed for modeling sequence similarity occurring in biological sequences. Next, we provide a high-level intuitive understanding of HMMs, profile HMMs and their use for sequence matching.

An HMM [23] is a statistical tool that captures certain properties of one or more sequences of observable symbols (such as NFS opcodes) by constructing a probabilistic finite state machine with artificial hidden states responsible for emitting those sequences. During training, the state machine's graph and its state transition probabilities are computed to best produce the training sequences. Later, the HMM can be used to evaluate whether a new unseen "test" sequence is "of the same kind" as the training data, with a score to quantify confidence in the match. The test sequence gets a higher score if the HMM has to traverse higher-probability edges in its state machine to produce that sequence. Thus, the HMM's state machine encodes the commonality among various opcode sequences of a given application workload by boosting the probabilities of the corresponding state transitions. It identifies a new workload by measuring how well its opcode sequence makes the HMM to make high-frequency transitions.

A profile HMM is a special type of HMM with states and a left-to-right state transition diagram specifically designed, as explained in Section 3.4.2, to efficiently remember symbol matches as well as tolerate chance mutations (i.e., inserts and deletes) in observed symbol sequences. Unlike a fully connected state graph of a traditional HMM, the profile HMM's left-to-right transition graph enables very fast $O(N)$ matching of a test sequence against known workload patterns.

In this paper, we consider two specific problems where existing sequence-matching techniques are applicable:

- *Workload identification*: we are told that samples are only from one workload but not told which one. Can we say which workload it is from?
- *Annotation*: we are told that distinct workloads ran sequentially one after another. Can we mark the boundaries when the workloads were switched?

In the following sections, we provide a more formal description of the HMM construct, including the concept of sequence alignment and how it is central to do approximate matching of large symbol sequences like opcode traces.

3.2 A Brief Review of HMMs

An HMM is defined by an alphabet Σ, a set of hidden states denoted by Z, a matrix of state transition probabilities A, a matrix of emission probabilities E, and an initial state distribution π. The matrix A is $|Z| \times |Z|$ with individual entries A_{uv}, which denotes the probability of transiting to state v from u. The matrix E ($|Z| \times |\Sigma|$) contains entries E_{ut}, which denotes the probability of emitting a symbol $t \in \Sigma$ while in hidden state u. Let λ be the model's parameters; these depend on Σ, Z, A, E and π and hence written as $\lambda = (\Sigma, Z, A, E, \pi)$. If we see a sequence X, an HMM can assign a probability to it as follows (assuming a model λ):

$$P(X|\lambda) = \sum_{z} \prod_{k} A_{z_k, z_{k+1}} E_{z_k, X_k}$$

The (inner) product terms arise from the probabilities of transition from one state (z_k) to another state (z_{k+1}) in the sequence of states under consideration whereas the (outer) sum of terms arises from having to sum all the possible ways of emitting the sequence X through all possible sequence of states. There is an iterative procedure based on expectation maximization algorithms for determining the parameters λ from a training set [23]. Popularity of HMMs stems from the fact that there are efficient procedures such as *(a) Viterbi algorithm* [23]) to compute the most probable state Z given a sequence X, i.e. compute Z to maximize $P(Z|X)$ *(b) forward and backward procedures* [23]

to compute the likelihood, $P(X)$ and *(c) Expectation Maximization procedures* [23] to learn the parameters, (A, E, π) given a dataset of independent and identically distributed sequences.

3.3 Problem Definition

At this point we can state the problem more formally as follows. Let $\{S_1, S_2, \ldots, S_r\}$ be a set of traces obtained by executing r times a particular workload, say W. The traces are different as they are obtained by executing the workload with different parameters; they may also be different due to some stochastic events in the system. The jth symbol s_{ij} of the sequence S_i is generated from the alphabet Σ of all possible opcodes. Let the sequence S_i be of length n_i, i.e the index j varies from 1 to n_i. We consider the task of constructing a model on these r sequences such that when presented with a previously unseen sequence, X, the model can infer whether X was generated by executing workload W.

3.4 Profile HMMs for identifying workloads

We will begin by recalling a few definitions related to sequence alignment. We will then discuss profiles and Profile HMMs, finally ending with a scheme for classifying workloads using them.

3.4.1 On Aligning Multiple Sequences

Let $S_i = s_{i1}s_{i2}\ldots s_{in_i}$ $(i = 1, 2)$ be two sequences of different lengths n_1 and n_2 generated from an alphabet Σ. An *alignment* of these two sequences is defined as a pair of new equal length sequences $S_i^* = s_{i1}^* \ldots s_{in}^*$ $(i = 1, 2)$ obtained from $S_1(S_2)$ by inserting "$-$" states in $S_1(S_2)$ to record differences in the two sequences. Let n be the length of S_1^* (which is also that of S_2^*) with $(n_1 + n_2) \geq n \geq max(n_1, n_2)$. We will call s_{1k} and s_{2l} as **matched** if for some j , $s_{1j}^* = s_{1k}, s_{2j}^* = s_{2l}$. On the other hand if $s_{1j}^* =$ "$-$",$s_{2j}^* = s_{2m}$ then we will say that there is a *delete* state in S_1 and *insert* state in S_2.

The **global alignment** problem is posed as that of computing two equal length sequences S_1^* and S_2^* such that the matches are maximized and insertions/deletions are minimized. This problem can be precisely formulated for suitably defined score functions and solved by dynamic programming based algorithms [20]. Global alignment is a good indicator of how similar two sequences are.

The problem of **local alignment** tries to locate two subsequences one from each string such that they are very similar. This problem can be formulated as that of finding two subsequences which are maximally aligned in the global sense for a suitably defined score function. It also admits a dynamic programming based algorithm [26] and can be solved exactly.

However both global and local alignment are defined for a pair of sequences. As mentioned before, our interest is in inferring similarities in more than two sequences. This will require the notion of multiple alignment, which generalizes the notion of alignment to more than two sequences. **Multiple alignment** is defined as the set $\mathcal{S} = \{S_1^*, S_2^*, \ldots, S_r^*\}$ where, as before, S_i^* is obtained from S_i by inserting "$-$" states so that the length of all the resulting r sequences are equal, say n. Multiple alignment can be visualized as a $r \times n$ matrix where each row consists of a specific string and each column corresponds to specific position in the alignment. Each matrix entry can take values in $\Sigma \cup$ "$-$". Multiple alignments are useful in detecting similar subsequences which remain conserved in sequences originating from the same family. Thus multiple alignment can decide the membership of a given new sequence with respect to a family represented by the multiple alignment. Figure 1 shows an alignment of ten traces of opcodes generated by an *edit* workload. Each symbol in the alignment represents a particular opcode. The alignment shows regions of high conservation where more than half of the symbols in the column are present. These conserved regions capture the similarity between the traces of this workload. When identifying a previously unseen trace generated by the same workload, it would be desirable to concentrate on checking that these more conserved columns are present.

One can extend the dynamic programming based solutions for the pairwise case to the problem at hand. Unfortunately they are prohibitively expensive, $O(n^r)$ in both time and space [13], and are not very practical for detecting large file operation sequences (100s to 1000s) typical in networked storage workloads.

3.4.2 Introduction to Profile HMMs

A *profile* is said to be a representation of a multiple alignment (such as that of multiple proteins that are closely related and belong to the same family). One can attribute the slight differences between family members to chance mutations, whose underlying probability distribution is not known. It has been empirically observed that HMMs are extremely useful in building profiles from biological sequences [6].

Profile HMMs: For modeling alignments, a natural choice for hidden states correspond to Insertions, Deletions and Matchings. In a Profile HMM, each insert state I_i and match state M_i has a nonzero emission probability of emitting a symbol, whereas the delete state D_i does not emit a symbol. The non-emitting states make Profile HMMs different from traditional HMMs. From an insert state, it is possible to move to the next delete state, continue in the same insert state or go to the next match state (Figure 2). Each diamond, circle, and square represents insert, delete and match states respectively. From each insert, delete or match state, the possible state transitions are as follows:

```
- G A G L L G G A A   G G G G - - A A C C   S S S S - - G G V V   G D D S WWS - - -   - - - - S WG - - G
- - - - - - - - - -   - - - - - - - - - -   - - - - - - - - - L   - - - - WW- - - -   - - - - - - G - - G
- L WWS S R R L L     - - G G - - L L C C   S S S S G G G G V V   S S S S WWS S L L   S S WWS S G - - G
- - WWS S R R L L     A A G G - - L L C C   S S S S G G G G V V   - - - S WWS S L L   S S WWS S G - - G
S S WWS S G G V V     A A G G L L L L C C   S S S S G G G G V V   - - - S WWS S L L   S S WWS S G - - G
- - WWS S R R L L     G G - - - - L L C C   S S S S G G G G V V   - - - S WWS S L L   S S WWS S G - - G
- S WWL L G G L L     G G G G - - - - C C   S S S S - - G G V V   G D D - - WG G - -   - - - - A A G - - G
- S WWL L G G L L     G G G G - - - - C C   S S S S - - G G V V   L D D - - WG G - -   - - - - S G G - - A
- S WWL L G G L L     G G G G - - - - C C   S S S S - - G G V V   G D D S WWS G - -   - - - - G G G - - A
- S WW- - G G - -     G G G G - - - - C C   S S S S - - G G V V   G D D S WWS G - -   - - - - G G G G A G
    + + + + + + + +   + + + +         + +   + + + +     + + + +         + + + + +             + + +     +
```

Figure 1. An example of multiple alignment of ten NFSv3 traces generated by an *edit* workload using the wireshark [5] tool. Here, G is getattr, S setattr, L lookup, R read, W write, A access, D readdirplus, C create, M commit, V remove, etc. Aligned columns are annotated at the bottom by a '+' if the opcodes in those columns are highly conserved. These columns will be modeled as match states in the profile HMM.

$$
\begin{array}{lll}
I_i & \rightarrow & D_{i+1}, \quad I_i, \quad M_{i+1}, \\
D_i & \rightarrow & D_{i+1}, \quad I_i, \quad M_{i+1}, \\
M_i & \rightarrow & D_{i+1}, \quad I_i, \quad M_{i+1}.
\end{array}
$$

Profile HMMs are essentially *Left-Right* HMMs (Figure 2). Unlike fully connected state machines, Left-Right HMMs have a more sparse transition matrix and are often upper triangular. Inference on such machines is much quicker and hence often preferred in many applications such as speech processing [23].

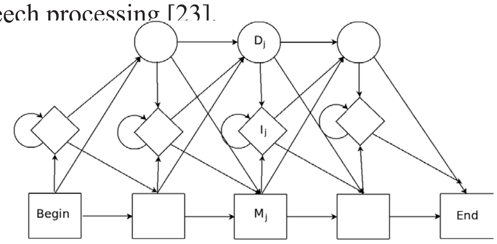


Figure 2. The transition structure of a profile HMM [8]. For example, from an insert state (diamond), we can go to the next delete state (circle), continue in the insert state (self loop) or go to the next match state (rectangle). Note that while multiple sequential deletions are possible by following the circle states, each with a different probability, multiple sequential insertions are only possible with the same probability.

It is straightforward to adapt the traditional HMM algorithms such as Viterbi algorithm, Forward-Backward procedure and Expectation Maximization based learning procedure [23] to profile HMMs [6, 8].

These models provide flexibility in modeling closely related sequences by the choice of more complex score functions. This has made profile HMMs extremely popular for comparing biological sequences.

Learning a Profile HMM from data: The parameters of profile HMMs are the emission probabilities and the state transition probabilities. This is easy to compute if one knows the multiple alignment. In such a case, the state transition probabilities are given by $a_{uv} = \frac{AN_{uv}}{\sum_v AN_{uv}}$ and the emission probabilities are given by $e_{ut} = \frac{EN_{ut}}{\sum_t EN_{ut}}$ where AN_{uv} denotes the number of transitions from the state u to v and EN_{ut} denotes the number of emissions of t given a state u(see [6]).

3.4.3 Profile HMM for identifying workloads

Let us now revisit the problem as defined in subsection 3.3. Assume that we have pretrained many Profile HMMs, each for a workload. Now consider the problem of identifying the underlying workload when a new trace is presented. Using Profile HMMs one can consider solving such a problem by the decision rule

$$y(X) = argmax_k P(X|\lambda_k)$$

where X is the unseen sequence, λ_k denotes the model for the kth workload and $y(X)$ is prediction for the underlying workload which generated the sequence X. Using the forward-backward procedure we can compute this decision rule easily. This can be understood as globally aligning the profile with the unseen sequence. Though there is no confidence measure with respect to prediction, the input is rejected (no prediction is made) if a confidence threshold is not crossed.

Now consider the problem of annotating a huge trace of opcodes generated by sequentially running workloads. As before assume that we have pretrained models of individual workloads. This would be equivalent to computing a local alignment of each profile with the bigger trace.

It is thus clear that the Profile HMM architecture chosen should be versatile enough to solve such problems. The architecture shown in Figure 2 will require some tweaking or the inference mechanism needs to be modified for such problems.

A Specific Implementation for Profile HMMs: For our work here, we have used the open source HMMER [7] implementation of a profile HMM whose architecture (Figure

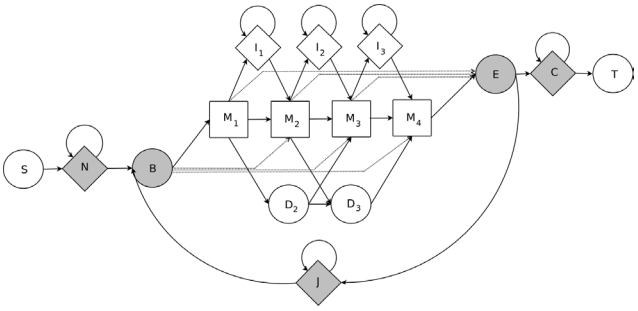


Figure 3. Architecture of HMMER [7]. Squares represent match states w.r.t. an alignment, diamonds are insert and ignored emitting states (N,J,C), circles are delete and special begin/end states (B,E,S,T). Note that there are no D to I or I to D transitions in HMMER.

3) allows flexibility in deciding between global and local alignments by adjusting the parameters of self-transitions involving nodes N (at the beginning), C (at the end), and J (in between). These self-transitions model the unaligned (or "ignored") part of the sequences. The set of states with their abbreviations are as follows:

M_x	Match state x, emitter.
D_x	Delete state x, non-emitter.
I_x	Insert state x, emitter.
S	Start state, non-emitter.
T	Terminal State, non-emitter.
N	N-terminal unaligned sequence state in the beginning of a sequence, emitter.
B	Begin state (for entering main model), non-emitter.
E	End state (for exiting main model), non-emitter.
C	C-terminal unaligned sequence state at the end of a sequence, emitter.
J	Joining segment unaligned sequence state, emitter

If the loop probability modeling the transition between $N \rightarrow N$ is set to 0, all alignments are constrained to start at the beginning of the model. If the probability of transition from $C \rightarrow C$ is set to 0, all alignments are constrained to end at the last node of the model. Setting $E \rightarrow J$ to 0 forces a global alignment. If it is not set to 0, the model can start at any point in a larger sequence and end some distance away for effecting local alignments. This option can be used for the sequence annotation task mentioned before by aligning the model locally against a large sequence. Furthermore, the transition $J \rightarrow J$ can be used to control the gap between local alignments. One can do the reverse, i.e., globally aligning a smaller sequence to a part of the model, by controlling the transitions between $B \rightarrow M$ and $M \rightarrow E$. HMMER is an extremely versatile and powerful sequence alignment tool. It can thus be very useful in locating sequences of opcodes from traces.

To learn the parameters of the model, it may be useful to use a small set of multiply aligned sequences. We have used an open source implementation of multiple alignment provided in [9] for this purpose.

3.5 Workload Identification Workflow: An Overview

In this section, we give an overview of our methodology using profile HMMs. Figure 4 gives the workflow for building a profile HMM model of a given workload. We need to supply one or more opcode sequences corresponding to traces of different runs of an application workload. These opcode sequences need to be encoded into a limited-sized alphabet that the HMM model works with. This is done by the alphabetizer module. The encoded sequences pass through a multiple alignment module (explained in Section 3.4.1), which creates a canonical aligned sequence for training. We use an open-source tool called Muscle [9] for this purpose. We then use HMMER [7] to generate a profile HMM model of the workload based on the aligned sequences.

To annotate the occurrences of a set of trained workloads in an arbitrary NFS trace, we extract the NFS opcode sequence from the trace, alphabetize it and pass it to the HMMER's pattern search tool called *hmmpfam* along with the profile HMM models of the workloads that we want to identify within the trace. The tool outputs the indices of the subsequences that it matched with various workloads along with a fractional score (in the range 0 to 1) indicating its confidence in the match relative to other workloads. We have written a script to post-process this output to produce the final annotation of the test sequence. The post-processing phase involves the following steps:

1. Merge two contiguous matches of the same workload.
2. Remove the matching subsequence with very low score (less than 0.1 percent of the average score for the matching subsequences of the same workload).
3. Again, merge any two new contiguous matching subsequences of the same workload.
4. If more than two workloads are reported for the same region, report the workload with a higher score.

4 Evaluation

In this section, we illustrate the capabilities of our profile HMM based methodology including its ability to identify and mark out the positions of high-level operations in an unknown network file system trace as well as its ability to isolate multiple workloads running concurrently. We also evaluate the training and pattern recognition performance of the methodology via micro-benchmarks.

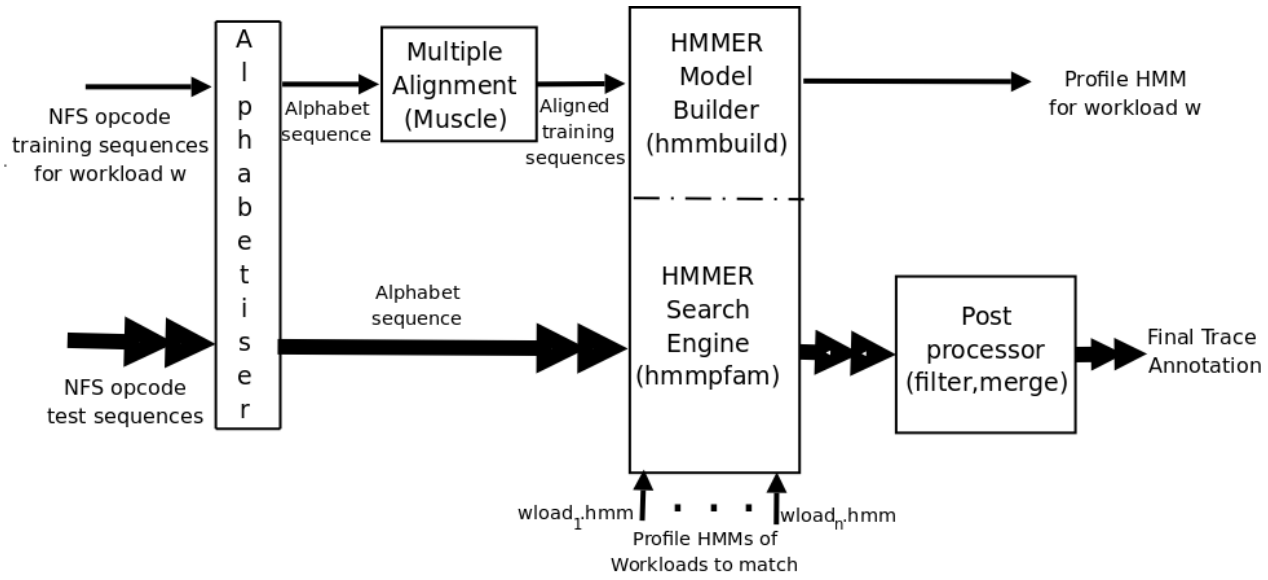


Figure 4. Profile HMM Training and usage workflow. Given a set of opcode traces of a given workload *w* with various parameters, this workflow produces a profile HMM model in the file *w.hmm*. Muscle and HMMER are existing open source tools, whereas the alphabetizer and post processor are modules that we developed. The bottom flow represents trace identification, where we input the workload models developed by the training workflow above into the HMMER search engine.

4.1 Experimental Setup and Training Method

For our evaluation, we choose several popular UNIX commands and user operations on files and directories as our application workloads: *tar, untar, make, edit, copy, move, grep, find, compile*. The UNIX commands access subsets of 14361 files and 1529 directories up to 7 levels deep stored on a Linux NFSv3 server from one or more Linux NFSv3 clients. For a more realistic evaluation, we also incorporated TPC-C [22] workloads. TPC-C is an OLTP benchmark portraying the activities of a wholesale supplier, where a population of terminal operators executes transactions against a warehouse database. Our TPC-C configuration used 1 to 5 warehouses with 1 to 5 database clients per warehouse. The database had 100,000 items.

The NFS clients are located on the same 1 Gbps LAN with NFS client-side caching enabled. The caching effects across multiple experiments were eliminated by mounting and unmounting the file system between each experiment. We capture the NFS packet trace at the NFS server machine's network interface using the Wireshark tool [5], and filter out the data portion of the NFS operations. For all experiments in this paper, we only use the opcode information in the NFS trace. Hence, we use the term *trace* in the rest of this section to refer only to the opcode sequences.

We build profile HMMs for each of the UNIX commands as follows. First, we run the UNIX command many times with different parameters and capture their traces. The number of captured traces for each command along with their average length in opcodes, is shown in Table 3. Next, we build the profile HMM for the command with increasing numbers of randomly selected traces as outlined in Figure 4, each time cross-validating its recognition quality by testing with the remaining traces. We stop when the improvement in the model quality metric diminishes below a threshold. We found that ten traces of each command were sufficient. We call those sequences as our *training sequences*, and the rest as *test sequences*.

4.2 Workload Identification

Our first experiment evaluates how well profile HMM can identify pure application-level workloads based on past training. We feed the test sequences to the trained profile HMM for identification. Table 3 shows the results in the form of a "confusion" matrix. Each row of the matrix indicates a test command and each column under the "models" umbrella indicates a command for which profile HMM got trained. Each cell indicates how well the profile HMM labeled the sequence as the given command, the ideal being 100%. Commands were recognized correctly much of the time with a few exceptions.

For instance, about 9% of the *copy* workloads are mislabeled as *edit* workloads. These were primarily single file copies and they share similarities with *edit* workloads that we trained with; they both exhibit an even mix of reads and writes. Copies of multiple files or recursive copies were not confused with *edit* workloads. The results also show that 11.3% of *grep* workloads are getting mis-labeled as *tar* workloads. Upon close inspection, we discovered that many

Trace Command	Models								
	make	find	grep	tar	untar	copy	move	edit	tpcc
make	**91.7**	1.2		1.2	2.4	3.6			
find		**91.8**	2.1			3.1	1		2.1
grep	1		**72**	22	5				
tar				**100**					
untar				1.2	**98.8**				
copy		1	1		6	**82**	1	9	
move		5.6	0.8	0.8		2.4	**89.6**	0.8	
edit								**100**	
tpcc									**100**

Table 3. Recognizing a single workload using the profile HMM on a test opcode sequence. Confusion matrix gives entries indicating the percentage of instances recognized correctly; the rows add up to 100%. The profile HMM recognized most commands correctly.

of the single-file *grep* commands ("grep foo bar.c") were being identified as *tar*'s. The combined multiple alignment model shows that the initial subsequence of *tar*, where a single file is being read from beginning to end, is very much like that of a single-file *grep*. That could have led to the profile HMM making an error. The diversity of the training set is critical. For instance, when we manually picked the *grep* training traces to have diverse command traces, we could improve the accuracy from 72% to 85%.

Consider another example: *find* and *tar* need to traverse a directory hierarchy in its entirety, except that in our case, *tar* additionally reads the file contents and writes the tar file. This distinction was enough for profile HMM to successfully distinguish *find* from *tar* in 100% of the cases. Overall, our methodology is able to distinguish workloads well based on small differences in their trace patterns.

An interesting result here is that the *tpcc* workload was identified correctly 100% of the time. The intuition behind this result is that, a complex workload contains unique patterns in its traces that can be accurately recognized. A simple workload may not have a strong signature in its traces, leading the profile HMM to mis-identify it occasionally.

Discrimination between TPC-C and Postmark: We also wanted to see how two large applications can be accurately distinguished using the NFS traces; we selected TPC-C and Postmark for this experiment. Postmark [15] is a synthetic benchmark that has been designed to create a large pool of continually changing files and measure the transaction rates for a workload approximating a large Internet electronic mail server.

Postmark traces were generated by running the benchmark 60 times with varying parameters. The file sizes were varied between 10000 bytes and 300000 bytes, the fraction of creations vs. deletions was varied between 10% and 100%, and the fraction of reads vs. appends was varied between 10% and 100%. Out of this set of traces, 10 were randomly picked for training, and 50 traces for testing. Similarly, 20 traces of previously unknown TPC-C workload were attempted after training with 4 traces. The TPC-C traces were from the previous experiment. The results of the workload identification are given in Table 4.

	TPC-C	Postmark
TPC-C	100%	0%
Postmark	0%	100%

Table 4. Workload identification accuracy with TPC-C and Postmark loads.

In both cases, there were no misclassifications. This experiment shows the capability of profile HMMs in discriminating between two complex and large workloads.

4.3 Trace Annotation

Our next experiment evaluates how profile HMM can mark out the NFS operations constituting various commands in a long but not earlier seen NFS packet trace. It tells us how accurately it can detect the start and end of commands just by observing the NFS operations. We run sequences of commands to simulate a variety of common user-level activities, collect their NFS opcode traces and query the profile HMM to identify the commands and their positions in each trace, as outlined in Figure 4. We then compare them with the known correct positions. Profile HMM is able to detect the boundaries of a command's opcode sequence to within a few opcodes in many cases.

Figure 5 shows the trace annotation diagram with both the detected and actual command boundaries for a command sequence <*untar;make;edit;make;tar*> that attempts to simulate the process of downloading the HMMER source package, compiling it, modifying it, compiling it again, and then *tar*'ing up the resulting package. The bottom-most bar in the figure shows the actual command boundaries, while the other bars show the annotation made by the profile HMM. We see that the quality of annotation is high. The NFS operations corresponding to the *untar*, the two *make*'s

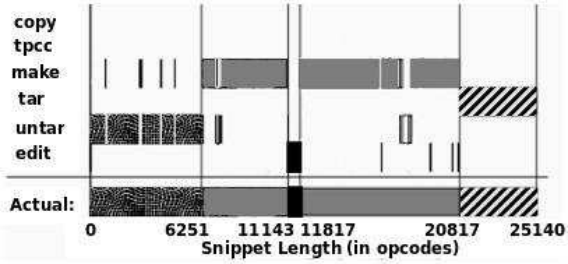


Figure 5. Visualization of the annotated trace for a sequence of user commands: <*untar*; *make*; *edit*;*make*; *tar*>. The bottom-most bar in the figure shows the actual sequence in the trace, while the other bars above show the annotation by the profile HMM. The vertical lines indicate workload transition boundaries. The visualizations in this figure show that the annotation is reasonably accurate. *make* is a harder command to classify because it invokes other commands.

and *tar* commands are accuratelv marked.

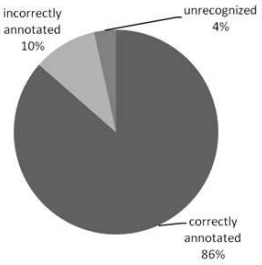


Figure 6. Overall Trace Annotation Accuracy for a random sequence of UNIX commands.

We then ran a comprehensive experiment, so that our results can be more statistically significant. We generated 100 traces, where each trace contained a run from a sequence of 100 commands, each picked randomly from our available pool of commands. We analyzed the traces using profile HMM, and annotated each opcode with its identified command. The results are presented in Figure 6. The annotation accuracy is a measure of how much of the trace is marked correctly with respect to start and end of the traces (and unrelated to confusion matrix entries computed for workload identification). 86% of the opcodes were annotated correctly; 10% of them were marked as belonging to a wrong command; and, 4% were identified as not belonging to any of our commands. Figure 7 shows the results broken down on a per-workload basis. Here we notice that opcodes belonging to *grep* and *move* were often incorrectly annotated. Both these workloads perform poorly in the sampling experiments above as well, implying that their characteristic patterns are not very unique.

In summary, profile HMMs are able to make use of subtle differences in workload traces to accurately identify transitions among workloads and annotate opcodes with the higher-level operations that they represent. The minor discrepancies observed were likely caused by not having

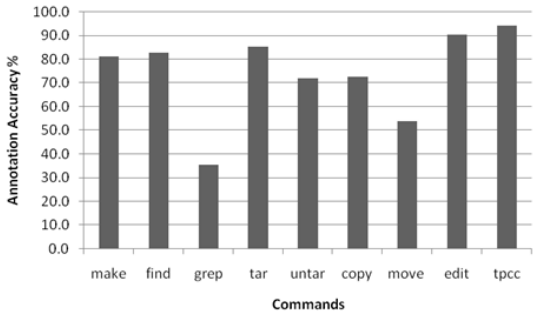


Figure 7. Trace Annotation Accuracy on a per-command basis. Note that it is lower than that for identification as the starting and ending of the traces have also to be marked correctly.

enough diversity in the selected training traces. Note that for single workload identification described in 4.2, manually picking the *grep* training traces to have diverse command traces resulted in accuracy improvement from 72% to 85%. Further work is needed to figure out how to select traces for improved discrimination.

4.4 Trace Processing Rate

Next, we measure the rate at which the profile HMMs can process (identify or annotate) a trace by applying it on a trace of length 50000 opcodes. Such a trace is constructed randomly using traces in our test sequence set. For identification, each model in turn reports how many instances of its family are present in the whole trace as well as a score that indicates how well it matches with its training set. For annotation, each model marks out its portion in the trace and a post-processing procedure decides which workload is assigned to a segment of the trace (based on a score).

Profile HMMs are not particularly fast – they processed the trace at a rate of 356 opcodes per second on a Intel Quad-Core CPU at 2.66 GHz and 3 GB of memory running Ubuntu Linux, kernel version 2.6.28. We then isolated each model and measured their performance individually on the same trace. The results are shown in the "processing rate" column of Table 5. We find that the models differ markedly in their speed (*make* and *tpcc* being the slowest). We see a strong inverse correlation between the speed of the model and the maximum sequence length of the training traces. This is understandable: shorter training sequences will likely build a profile HMM with fewer states and transitions. One could speed up the models by choosing shorter traces for training, provided they do not jeopardize the identification accuracy. This is a tradeoff worth exploring in the future.

Trace Command	# Test Traces	Trace Length			Processing rate (opcodes/sec)
		min.	mean	max	
make	84	23	2653	32175	2971
find	98	33	10683	66093	135893
grep	100	19	4784	24024	121701
tar	98	67	1255	19578	49430
untar	81	85	2082	28013	24680
copy	100	35	8665	97789	21408
move	125	9	26	39	667714
edit	127	657	670	687	22177
tpcc	24	1289	12665	61430	565

Table 5. Trace processing rates. Since each model has different number of states in its profile HMM, the processing rates differ.

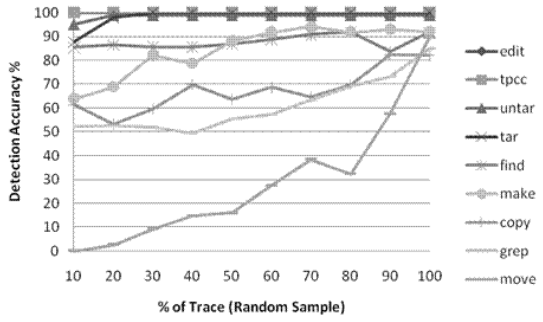


Figure 8. Sensitivity of profile HMM to the length of the trace sample analyzed for various commands when sample picked randomly from the whole trace. Y-axis indicates the percent of runs (out of hundred runs) where the command was correctly recognized.

4.5 Identification of Randomly Sampled Partial Traces

In a real system, we will not have the entire trace of a single command or a neatly ordered sequential set of commands to analyze. They will typically be interleaved because of concurrent execution. Therefore, we must be able to detect an application operation just by observing a snippet of a command's trace. Further, for online behavior detection and adaptation, we should be able to quickly detect an application operation, which implies that we should need to analyze small amounts of traces to identify workloads.

Our next experiment evaluates how much of a randomly sampled NFS trace the profile HMM methodology needs to be able to correctly recognize a high-level operation. For this experiment, we feed the profile HMM with contiguous substrings of the pure test sequences — of various lengths and at random locations in the full sequence — and measure how often it detects the command correctly. Figure 8 contains plots of profile HMM's sensitivity to trace snippet size for various high-level commands. As the graphs indicate, profile HMM is able to recognize most workloads with

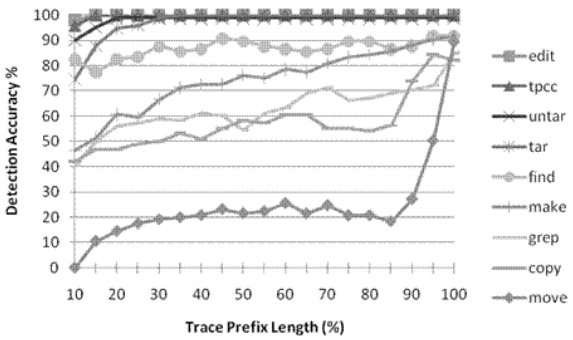


Figure 9. Sensitivity of profile HMM's accuracy to the length of the trace prefix analyzed for various commands. The Y-axis indicates the percent of runs (out of hundred runs) where the command was correctly recognized.

80% accuracy by examining a small fraction of the trace. The *move* command generates a small trace to begin with. Therefore, the profile HMM requires a large fraction of its trace to be examined to correctly identify it.

The characteristic patterns of a workload may be concentrated at some locations for certain commands, while they may be distributed better for other commands. Having characteristic patterns at various locations in the trace is useful for online behavior detection, since there is a larger likelihood of identifying a workload from a random sample. To understand the distribution of characteristic patterns in our workloads, we tested the profile HMM with varying length prefixes of traces. Figure 9 shows the results. We see that the predictive value of small prefixes of traces is quite high. For some commands like *copy* and *move*, the end of a trace seems to have strong characteristics.

This evaluation suggests that in real scenarios, some workloads may be identified by examining just a small snippet, while other workloads may need a large fraction of their traces to be analyzed before identification.

4.6 Automated Learning on Real Traces

Validating our approach using real traces from real deployments is important. Our approach is based on a classification-based methodology that requires that the training data be labeled. Unfortunately, real traces are typically not labeled with workload information. Therefore, we will neither be able to train with the real trace nor be able to validate our results.

To tackle this problem, we use the LD_PRELOAD environment variable on the client to interpose our own library that intercepts all process invocations ("exec" family of calls in UNIX) and forces a sentinel marker in the trace by doing an operation that can be spotted. Whenever we see an "exec", we "stat" a non-existent file – the file name encodes the identity of the exec'ed program. The NFS re-

	gcc	cat	mv	ld
gcc	80.5	1.9	0.9	16.8
cat	3.1	77.9	0.8	18.2
mv	0.6	0.5	62.5	36.4
ld	13.3	1.2	1.7	83.8

Table 6. Workload identification accuracy on live traces.

sponse that the file does not exist (ENOENT) with the coded filename is enough for us to mark the boundaries of the trace segment generated by each of the command invocations. Here we need to ensure that the invocation is "atomic", i.e., it does not result in exec'ing of other programs that are of interest independently for identification (otherwise, we will mark a only a subtrace as belonging to the invocation and mark some part of the following trace as belonging to the subprocess). We used an open-source tool called *Snoopy* [21] and modified it to suit our purposes.

As an example, we used the compilation of Linux 2.6.30 source as the generator of a real trace. We instrumented the client with the above interposition library, collected the traces for a certain amount of time and constructed our training trace data automatically. Our sentinel markers in the trace also give us an easy way to validate our results.

The following commands were detected in the Linux source compilation on the Ubuntu 9 system[1]: "gcc", "rm", "cat", "mv", "expr", "make", "getent", "cut", "mkdir", "bash", "run-parts", "sed", "date", "whoami", "hostname", "dnsdomainname", "tail", "grep", "cmp", "sudo", "objdump", "ld", "nm", "objcopy", "awk", "update-motd", "renice", "ionice", "basename", "landscape-sysinfo", "who", "stat", "apt-config", "ls". Since commands like "make" initiate, for example, many gcc compiles, it is not possible to demarcate the beginning and end of the trace that "make" contributes as we are interested in "gcc" as a workload in itself. We eliminated such composite commands and those that do not contribute to NFS traces (eg. "date"), and ended finally by selecting 4 commands in the live trace.

For workload identification, we considered the 105 minute live trace of the Linux source compilation discussed earlier with training on approximately 3 minutes of the trace. The results are given in Table 6.

To understand how learning is improved with larger number of training traces used, we chose 30 sec, 40 sec, 50 sec, 1 min, 2 min, 3 min, 4 min and 5 min durations of the trace and used the specific workload found in these durations for training that workload. From Figure 10, we notice that the accuracy of the workload identification improves with increase in the number of training sequences used, thus demonstrating learning in the system. Commands that generate a small amount of traces, such as *cat* and *mv* pose difficulties for our methodology. In this experiment, the output of the *cat* commands were for */dev/null* and for a single specific file; because of client-side caching, the traces did not have a strong signature. We need traces with good signatures (like *gcc*) to get good results. This is acceptable from a practical standpoint as bigger application workloads, in general, are of more interest in the systems community.

[1] "landscape-sysinfo" provides a quick summary about the machine's status regarding disk space, memory, processes, etc. "run-parts" runs a number of scripts or programs found in a single directory.

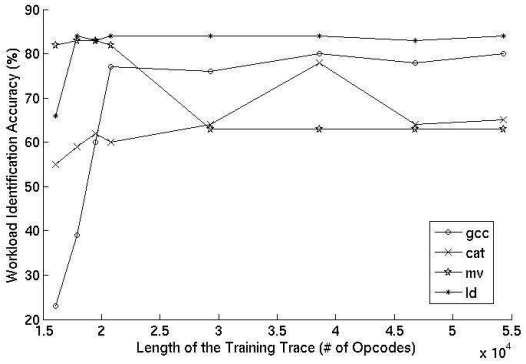


Figure 10. Online learning on live traces.

The value of the profile HMM as a practical tool will be significantly enhanced if we can automatically generate a labeled trace, with each of its constituent workloads demarcated, for training. The LD_PRELOAD mechanism is a way to do this. On new clients or clients running new applications, the interposition library could be introduced to generate new training sets. The library could subsequently be removed after sufficient training data has been generated.

4.7 Concurrent Workloads

Shared storage systems almost always serve multiple concurrent workloads. Therefore, the server-side trace contains the trace sequences of multiple application-level operations interleaved with each other in time. However, while a shared storage system may serve files to thousands of clients in an enterprise deployment, the NFS trace contains client IDs that can be used to tease the interleaving apart. Therefore, we need automated tools only to separate out the traces due to requests from a single client. Typically, the number of concurrent applications at a single client invoking NFS operations to the same backend server are small.

Profile HMM's ability to detect high-level commands from small snippets of file system operations helps identify the various workloads running concurrently. Our next experiment evaluates this ability. We run sequences of commands from 2 to 6 NFS clients accessing the same NFS server, capture the NFS opcode trace at the server's network interface, remove the client ID (to simulate the effect of multiple applications from the same client), and feed it into the profile HMM for marking the commands' opera-

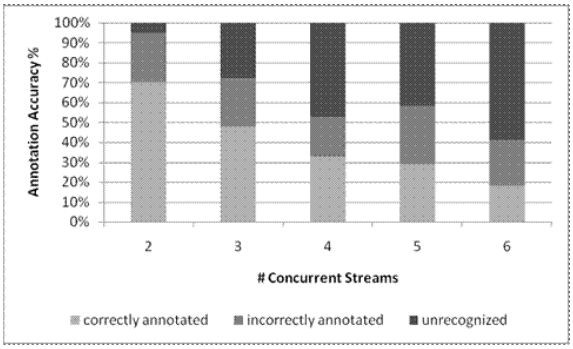


Figure 11. Concurrent sequences of commands were run from 2 to 6 clients. The graph shows the quality of the annotation.

tion sequences. We compare the result with the sequences identified manually based on the source IP address. Figure 11 shows the quality of the annotation. The amount of concurrency determines whether there will be long enough snippets for profile HMM to accurately annotate the trace. As expected, for a concurrency level of 2 or 3, the results are acceptable, but gets worse beyond that. The interesting point to note here is that the incorrect annotations do not increase with concurrency; only the proportion of unrecognized sequences do. The profile HMM's ability to explicitly tag unrecognized sequences as such helps the user rely on its output.

More than the exact marking of regions, the identification of constituent workloads in a mixed-workload scenario is itself of good value. This is because, for the typical administrator, a more compelling use case than unraveling the opcode sequences of interleaving workloads is to identify which workloads are running in a given interval of time. Note that TPC-C, a very concurrent workload, can be identified quite successfully as reported earlier (Sections 4.2, 4.3).

5 Limitations

During the course of our evaluation, we discovered a few limitations with this methodology. First, training the tool requires a diverse and representative sample of workloads. This is a fundamental characteristic of machine learning methodologies. Second, the open-source tools that we used to build our solution are from computational biology. The current off-the-shelf solutions have a limited alphabet space which may not be completely appropriate for systems applications. However, we believe that there are no fundamental mathematical limitations in the number of symbols, except that we may have to perform significantly more training if we use more symbols. Third, the level of concurrency at a client adversely affected the accuracy of the tool. The fine-grained interleaving resulting from a large number of concurrent streams can be tackled only if we are able to identify workloads using very small trace snippets. Finally, the profile HMM seems to be slow compared with the typical rates of NFS operations at a server, hampering online analysis. Many of these limitations may not be fundamental in nature, but pointers to future work.

6 Conclusions and Future Work

In this paper, we have presented a profile HMM-based methodology for analysis of NFS traces. Our method is successful at discovering the application-level behavioral characteristics from NFS traces. We have also shown that given a long sequence of NFS trace headers, it is able to annotate regions of the sequence as belonging to the applications that it has been trained with. It can identify and annotate both sequential and concurrent execution of different workloads. Finally, we demonstrate that small snippets of traces are sufficient for identifying many workloads. This result has important consequences. Because traces are going to get generated faster than one can analyze them, being able to infer meaningful information from periodic random sampling is very important for effective analysis.

Although profile HMM methodology looks promising for trace analysis, our experience indicates that we have not leveraged all its capabilities. For instance, we have not used all the information that is available in the NFS trace. There is a rich amount of data available in the form of file names and handles, file offsets, read/write lengths and error responses that throw more light on the application workloads. We have to investigate how to incorporate this information into a form amenable for multiple alignment and profile HMM. This will be the first step in extending our work.

NFSv4 introduces client delegations, offering clients the ability to access and modify a file in its own cache without talking to the server. This implies that an NFSv4 trace may not have all the information about application workloads. Investigating how profile HMMs work on NFSv4 traces is a clear extension of this work.

We also believe that our methodology is general enough that we can apply it to other source data such as network messages, system call traces, disk traces and function call graphs. This methodology can be a foundation to tackle use cases in areas such as anomaly detection and provenance mining, which are building blocks for next-generation systems management tools. Finally, we will look into other machine learning methods that overcome some of the limitations of profile HMMs.

Acknowledgments: We thank Bhupender Singh, Alex Nelson, and Darrell Long for reviewing the paper, Pavan Kumar for performing the PostMark experiments, and Alma Riska for shepherding the paper with thoughtful comments and guidance. We also gratefully acknowledge support

from a NetApp[2] research grant.

References

[1] Eric Anderson. Capture, conversion, and analysis of an intense NFS workload. In *Proccedings of the 7th conference on File and storage technologies,* pages 139-152, Feb. 2009

[2] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout. Measurements of a distributed file system. In *Proceedings of the 13th Symposium on Operating Systems Principles*, Oct. 1991.

[3] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *Proc of the Seventh Symposium on Operating System Design and Implementation*, pages 259–272, Dec. 2004.

[4] B. Callaghan, B. Pawlowski, and P. Staubach. NFS version 3 protocol specification. Internet Request For Comments RFC 1813, Internet Network Working Group, June 1995.

[5] G. Combs. Wireshark network protocol analyzer. `http://www.wireshark.org`, 1998.

[6] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic models of proteins and nucleic acids*. Cambridge University Press, 1998.

[7] S. R. Eddy. HMMER: Sequence analysis using profile hidden Markov models. Available at `http://hmmer.wustl.edu/`.

[8] S. R. Eddy. Profile hidden Markov models. *Bioinformatics*, 14(9):755–763, 1998.

[9] R. C. Edgar. MUSCLE:multiple sequence alignment with high accuracy and high throughput. *Nucleic Acid Research*, 32(5):1792–1797, 2004.

[10] D. Ellard. *Trace-based analyses and optimizations for network storage servers*. PhD thesis, Cambridge, MA, USA, 2004. Adviser-Margo I. Seltzer.

[11] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS tracing of email and research workloads. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST03)*, pages 203–216, 2003.

[12] D. Ellard and M. Seltzer. New NFS tracing tools and techniques for system analysis. In *Proceedings of the Seventeenth Large Installation Systems Administration Conference (LISA)*, Oct. 2003.

[13] D. Gusfield. *Algorithms on Strings, Trees and Sequence*. Cambridge University Press, 1997.

[14] D. Haussler, A. Krogh, I. S. Mian, and K. Sjölander. Protein modeling using hidden markov models: analysis of globins. In *Proceedings of the 26th Annual Hawaii International Conference on Systems Sciences*, volume 1, pages 792–802. IEEE Computer Society, 1993.

[15] J. Katcher. Postmark: A new file system benchmark. Technical Report 3022, NetApp, 1997.

[16] A. Krogh, M. Brown, I. S. Mian, Sjölander, and D. Haussler. Hidden Markov models in computational biology: Applications to protein modeling. 235:1501–1531, 1994.

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

[18] T. Madhyastha and D. Reed. Input/output access pattern classification using hidden markov models. In *Workshop on Input/Output in Parallel and Distributed Systems*, Nov. 1997.

[19] M. Mesnier, E. Thereska, G. Ganger, D. Ellard, and M. Seltzer. File classification in self-* storage systems. In *Proceedings of the First International Conference on Autonomic Computing*, May 2004.

[20] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970.

[21] D. Packages. Snoopy. `http://http://packages.debian.org/lenny/snoopy`.

[22] F. Raab, W. Kohler, and A. Shah. Overview of the TPC benchmark C: The order-entry benchmark. `http://www.tpc.org/tpcc/detail.asp`.

[23] L. R. Rabiner. Tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of IEEE*, 77(2):257–288, 1989.

[24] D. Roselli, J. Lorch, and T. E. Anderson. A comparison of file system workloads. In *Proceedings of the USENIX 2000 Annual Technical Conference*, 2000.

[25] R. R. Sambasivan, A. X. Zheng, E. Thereska, and G. Ganger. Categorizing and differencing system behaviours. In *Hot Topics in Autonomic Computing*, June 2007.

[26] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:197–198, 1981.

[27] N. Tran and D. Reed. Automatic ARIMA time-series modeling for adaptive i/o prefetching. *IEEE Transactions on Parallel and Distributed Systems*, 15(4):362–377, Apr. 2004.

[28] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of ACM*, 21(1):168–173, 1974.

[29] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, May 1999.

[2] NetApp, the NetApp logo, and Go further, faster, are trademarks or registered trademarks of NetApp, Inc. in the United States and/or other countries.

Provenance for the Cloud

Kiran-Kumar Muniswamy-Reddy, Peter Macko, and Margo Seltzer
Harvard School of Engineering and Applied Sciences

Abstract

The cloud is poised to become the next computing environment for both data storage and computation due to its pay-as-you-go and provision-as-you-go models. Cloud storage is already being used to back up desktop user data, host shared scientific data, store web application data, and to serve web pages. Today's cloud stores, however, are missing an important ingredient: ***provenance****.*

Provenance is metadata that describes the history of an object. We make the case that provenance is crucial for data stored on the cloud and identify the properties of provenance that enable its utility. We then examine current cloud offerings and design and implement three protocols for maintaining data/provenance in current cloud stores. The protocols represent different points in the design space and satisfy different subsets of the provenance properties. Our evaluation indicates that the overheads of all three protocols are comparable to each other and reasonable in absolute terms. Thus, one can select a protocol based upon the properties it provides without sacrificing performance. While it is feasible to provide provenance as a layer on top of today's cloud offerings, we conclude by presenting the case for incorporating provenance as a core cloud feature, discussing the issues in doing so.

1 Introduction

Data is information, and as such has two critical components: what it is (its contents) and where it came from (its ancestry). Traditional work in storage and file systems addresses the former: storing information and making it available to users. Provenance addresses the latter. Provenance, sometimes called lineage, is metadata detailing the derivation of an object. If it were possible to fully capture provenance for digital documents and transactions, detecting insider trading, reproducing research results, and identifying the source of system break-ins would be easy. Unfortunately, the state of the art falls short of this ideal.

Current research has demonstrated the feasibility of automatically capturing provenance at all levels of a system, from the operating system [18, 30] to applications [27]. Our goal is to extend provenance to the cloud.

Provenance is particularly crucial in the cloud, because data in the cloud can be shared widely and anonymously; without provenance, data consumers have no means to verify its authenticity or identity. The web has taught us that widely shared, easy-to-publish data are useful, but it has also taught us to be skeptical consumers; it is impossible to know exactly how updated or trustworthy data on the web are. We should solve the problem now while cloud services are still new and evolving. For example, Amazon's "Public Data Sets on AWS" provides free storage for public data sets such as GenBank [2], US census data, and PubChem [1]. If researchers are to make the most of these data sources, they must be able to accurately identify the process used to generate the data. Provenance, bound to the data it describes, provides the necessary information for verifying the process used to generate the data. Similarly, provenance can be used to debug experimental results and to improve search quality. We discuss these use cases in Section 2.2.

As both automatic provenance collection and cloud storage are relatively new developments, it is not obvious how to best record provenance in the cloud. We begin by identifying four properties crucial for provenance systems. First, *provenance data-coupling* states that when a system records data and provenance, they match – the provenance accurately describes the data recorded. Second, *multi-object causal ordering* states that ancestors described in an object's provenance exist, i.e., the objects from which another object is derived. This ensures that there are no dangling provenance pointers. Third, *data-independent persistence* states that provenance must persist even after the object it describes is removed. Fourth, *efficient query* states the system supports queries on provenance across multiple objects. We discuss these properties and the implications of violating them in Section 3.

Using these properties as a metric, we designed three alternative protocols for storing provenance using current cloud services. The protocols vary in complexity, the guarantees they make, and the distributed cloud components they involve. The first protocol is the simplest and uses only a cloud store. In turn, it is the weakest of the protocols. The second protocol satisfies a larger subset of the properties and uses a cloud store and a cloud database. The third protocol uses a cloud store, a cloud database, and a distributed cloud queuing service and satisfies all the properties. The database and

queue have the same availability, reliability, and scalability properties as the store. We discuss the protocols and the properties they satisfy in Section 4.3. We use a Provenance Aware Storage System (PASS) [30] augmented to use Amazon Web Services (AWS) [5] as the backend to build and evaluate the protocols for storing provenance. Based on our experience designing and implementing protocols for storing provenance on current cloud offerings, we discuss research challenges for providing native provenance support on the cloud.

The contributions of this paper are:

1. Definition of properties that provenance systems must exhibit.
2. Design and implementation of three protocols for storing provenance and data on the cloud, evaluating each protocol with respect to the properties we established.
3. Evaluation and comparison of the cost and performance of our three provenance storage protocols.

The rest of the paper is organized as follows. In the next section, we provide background on provenance and our provenance collection substrate, discuss use cases for provenance in the cloud, and introduce the cloud services that are most pertinent to this work. In Section 3, we present the desirable properties for storing provenance in the cloud. In section 4, we discuss the challenges unique to storing provenance on the cloud and present the architecture and implementation of our three provenance recording protocols. In section 5, we evaluate the protocols for overhead, throughput, and cost. We discuss related work in section 6. We discuss the challenges for providing native support for provenance in the cloud in section 7, and we conclude in section 8.

2 Background

Provenance can be abstractly defined as a directed acyclic graph (DAG). The DAG structure is fundamental and holds for all provenance systems irrespective of the software abstraction layer at which they operate. The nodes in the DAG represent objects such as files, processes, tuples, data sets, etc. The edges between two nodes indicates a dependency between the objects. Nodes can have attributes. For example, a process node has attributes such as the the command line arguments, version number, etc. A file node has name and version attributes. Each version of a file or process is represented by a distinct node in the DAG. The provenance graph, by definition, is acyclic as the presence of cycles would indicate that an object was its own ancestor.

2.1 Provenance Aware Storage System (PASS)

We use our PASS [30] system to collect provenance. PASS is a storage system that transparently and automatically collects provenance for objects stored on it. It observes application system calls to construct the provenance graph. For example, when a process issues a `read` system call, PASS creates a provenance edge recording the fact that the process depends upon the file being read. When that process then issues a `write` system call, PASS creates an edge stating that the file written depends upon the process that wrote it, thus transitively recording the dependency between the file read and the file written. For processes, PASS records several attributes: command line arguments, environment variables, process name, process id, execution start time, the file being executed, and a reference to the parent of the process. For all other objects (files, pipes, etc.), PASS records the name of the object (pipes do not have names). Prior to this work, PASS used local file systems and network attached storage as its storage backend; this work leverages PASS as a provenance collection substrate and extends its reach to using the cloud as the storage backend.

2.2 Cloud Provenance Use Cases

The following use cases illustrate the utility and need for provenance in the cloud.

Debug Experimental Results: The Sloan Digital Sky Survey (SDSS) [20] is an online digital astronomy archive consisting of raw data from various sources (e.g., imaging camera, photometric telescope, etc.). It also provides an environment for researchers to process and store data in personal databases. Since researchers use of the environment is bursty, one can imagine using cloud stores and virtual machines to provide this service. Consider a scenario where SDSS administrators upgrade the software distribution on the compute node images unbeknownst to the users. Suppose further that when users run their scripts, the resulting output is flawed. Without provenance, users are left to manually search for clues explaining the change in behavior. With provenance, users can compare the provenance of newly generated output with the provenance of older output to determine what has changed between invocations. For example, if a new JVM had been introduced, the difference in JVMs would be readily apparent in the provenance output.

Detect and Avoid Faulty Data Propagation: The SDSS processed data is produced by a pipeline of data reduction operations. A scientist using the data might want to ensure that she is using an appropriately calibrated data set. Without provenance, the scientist has no means to verify that she is using data processed by

the correct software. With provenance, the scientist can examine the data's provenance to verify that appropriate versions of the tools were used to process the data. In addition, provenance enables users to discover how far faulty data has propagated throughout a data processing pipeline.

Improving Text Search Results: Shah et. al. [39] showed that provenance can improve desktop search results. The provenance graph provides dependency links between files, similar to hyperlinks between webpages, that can be used to improve the quality of search results. Shah's scheme first uses a pure content-based search to compute an initial set of documents. Then, they traverse the provenance DAG of the initial document set P times. At each iteration of the traversal, they update the weight for each node based on the number of incoming/outgoing edges. After P runs, they re-rank the files and include new files to the list based on the weights computed.

Similarly, provenance can be used to improve search quality for data stored on the cloud. For example, consider a scenario where a user archives data on the cloud. Without any content-based indexing, searching that archived data requires downloading each file to the user's desktop. Content-based indexing reduces the number of files the user needs to download. Content-based indexing refined by provenance, such as inter-file dependencies, inputs, or command-line arguments from the program that generated the data, further reduces the effort required to locate a particular file.

2.3 Cloud Services

We next provide a brief description of the cloud services that are most pertinent to this work.

Object Store Service: A cloud object service allows users to store and retrieve data objects. Service providers generally provide a REST-based interface for accessing objects, with each object identified by a unique URI. The service allows users to *PUT*, *GET*, *COPY*, and *DELETE* objects. The PUT operation overwrites any previous versions of an object. With each object, clients can store some metadata, represented as <name,value> pairs. The PUT operation supports atomic updates to both data and metadata. The cost of using such services is based on the number of bytes transferred (both to and from), the storage space utilization, and the number of operations performed. Amazon *Simple Storage Service* (S3) [37] and Microsoft Azure *Blob* [6] are examples of object store services.

Database Service: A cloud database service provides index and query functionality. The data model is semi-structured, i.e., it consists of a set of rows (called items), with each row having a unique itemid and each item having a set of attribute-value pairs. The attribute-value pairs present in one item need not be present in another, and an item can have multiple attributes with the same name. For example, an item can have two phone attributes with different values. The database service provides the same reliability and availability guarantees as the data store. Amazon's *SimpleDB* [38] and Microsoft Azure's *Table* [8] are examples of such services. SimpleDB supports attribute names and values up to 1 KB, while Azure allows them to be up to 64KB. SimpleDB provides a traditional SELECT query interface, whereas Azure provides a LINQ [25] query interface.

Messaging Service: Distributed messaging systems provide a queuing abstraction allowing users to exchange messages between distributed components in their systems. Queues are typically identified by a unique URL. Users can perform operations such as *SendMessage*, *ReceiveMessage*, and *DeleteMessage*. The messaging service provides similar guarantees to that of the corresponding cloud store. Message delivery is generally best-effort, in-order message delivery. Amazon's *Simple Queueing Service* (SQS) [41] and Microsoft Azure *Queue* [7] are examples of such Messaging systems. Both SQS and Queue enforce an 8KB limit on messages.

2.3.1 Eventual Consistency

As with other distributed systems, building highly scalable cloud services involves making various choices in the design space. A number of recent systems that operate at the cloud scale have chosen to be provide high performance and high availability while providing a weaker form of data consistency, called eventual consistency. AWS is an example of an eventually consistent service suite. This implies that, for example, a client performing a GET operation on an S3 object immediately after a PUT on that object might receive an older copy of the object as S3 might service that request from a node that has not yet received the latest update. If two clients update the same object concurrently via a PUT, the last writer wins, but for a non-deterministic period of time after a PUT, a subsequent GET operation might return either of the two writes to the client. Azure services, on the other hand, are strictly consistent; a client is guaranteed to receive the latest version of an object. Eventual consistency dictates that clients must design appropriate mechanisms to detect inconsistencies between objects. We designed our protocols assuming eventual consistency, as it is the weaker form of concurrency; anything that works with eventual consistency will work trivially with stronger models.

3 Provenance System Properties

There are four properties of provenance systems that make their provenance truly useful. We motivate and introduce these properties.

Provenance Data Coupling The *data-coupling* property states that an object and its provenance must match – that is, the provenance must accurately and completely describe the data. This property allows users to make accurate decisions using provenance. Without data-coupling, a client might use old data based on new provenance or might use new data based on old provenance. In both of these cases, the user relying on the provenance is misled into using invalid data.

Systems that do not provide data-coupling during writes can detect data-coupling violations on access and withhold or explicitly identify objects without accurate provenance. For example, if the provenance includes a hash of the data, we can compute the hash of a data item to determine if its provenance refers to this version of that data. Detection is, at best, a mediocre replacement for data-coupling, because although users will not be misled, they cannot safely use available data when its provenance is wrong.

Given the eventual consistency model of existing cloud services and the fact that we cannot modify existing cloud services, we find a weaker form of the property, *Eventual data-coupling* practical. In eventual data-coupling, the data and its provenance might not be consistent at a particular instant, but are guaranteed to be eventually match. With eventual data-coupling, a system requires detection, since there may exist intervals during which an object and its provenance do not match.

Multi-object Causal Ordering This property acknowledges the causal relationship among objects. If an object, O, is the result of transforming input data P, then the provenance of O is the super-set of the provenance of P. Thus, a system must ensure that an object's ancestors (and their provenance) are persistent before making the object itself persistent. Multi-object Causal Ordering violations occur when the system writes an object to persistent store before writing all its ancestors, and the system crashes before recording those ancestors and their provenance. These violations produce dangling pointers in the DAG. Similar to eventual data-coupling, a weaker form of the property *Eventual Causal Ordering* is realizable. A system still requires detection to account for the intervals during which an object's provenance may be incomplete, because its ancestors and their provenance are not yet persistent or not available due to eventual consistency.

Data-Independent Persistence This property ensures that a system retains an object's provenance, even if the object is removed. As in the last section, assume that P is an ancestor of O. If P were removed, O's provenance still includes the provenance of P, so a system must make sure to retain P's provenance, even if P no longer exists. If P's provenance is deleted when P is deleted, parts of the provenance DAG will become disconnected. If P had no descendants, then a system might choose to remove its provenance, since it would no longer be accessible via any provenance chain. Another approach to solving this problem is to copy and propagate an ancestor's provenance to its descendants. This is inefficient in terms of space and can quickly become unwieldy.

Efficient Query Since provenance is created more frequently than it is queried, efficient provenance recording is essential. However, efficient query is also important as provenance must be accessible to users who want to access or verify provenance properties of their data. In scenarios where the number of objects are few or users already know the objects whose provenance they want to access, efficiency is not an issue. Efficiency matters, however, when the number of objects is sizeable and users are unsure of the objects they want to access. For example, users might want to retrieve objects whose provenance matches certain criteria. In scenarios such as this, if a system stores provenance, but that provenance is not easily queried, the provenance is of reduced value.

4 Protocol Design and Implementation

We begin this section by presenting the challenges unique to the cloud that guided our protocol design. Next, we present a high level architectural overview and implementation of our system. Finally, we describe each of our three protocols in detail. For each protocol, we discuss its advantages and limitations. For the rest of the paper, we use AWS as the cloud backend as it is the most mature product on the market.

4.1 Challenges

The cloud presents a completely different environment from the ones addressed by previous provenance systems. The cloud is designed to be highly available and scalable. None of the existing provenance solutions, however, account for availability or scalability in their design. The cloud is also not extensible, while all existing solutions required making changes to the operating system, the workflow engine, the application, or some other piece of software. Further, the long latency between users and the cloud presents different update and error models. These properties make managing provenance in the cloud different from managing it on local storage.

Extensibility: Most existing provenance systems assume the ability to modify system components. For ex-

ample, PASS uses either a file system or an NFS service as the storage backend. PASS defined new extensions to the VFS interface to couple data and provenance [28]. The Virtual Data Grid [17] and myGrid [42] workflow engines integrate provenance collection into the workflow execution environment. The PASOA [34] framework for recording provenance in service oriented architectures assumes the existence of a custom designed provenance recording service. In the case of the cloud, however, modifying or extending existing services is not possible.

Availability: One can imagine building a wrapper service that acts as a front to the cloud services and provides a cloud provenance storage service that satisfies the properties we identified. For the approach to be viable, however, the wrapper service has to match the availability of the cloud. If not, the overall availability is reduced to the availability of the wrapper service. Building such a highly available wrapper service is counterproductive as it requires a great deal of effort and infrastructure investment, defeating the very purpose of moving to the cloud. Hence, we design protocols that leverage existing services while satisfying the properties.

Scalability: In order to make the provenance queryable, most systems store provenance in a database. Hence, we considered storing the provenance in a database backed by an S3 object (e.g., a MySQL or Berkeley DB database stored in the S3 object). The provenance would then be queryable, but this approach would not scale. First, to avoid corrupting the database, clients need to synchronize updates between each other. A single global lock is a scalability bottleneck, and a distributed lock service would introduce the potential for distributed deadlock. Second, due to the update granularity of cloud stores, clients need to download the database object for every update, which also does not scale. One can, of course, use more sophisticated parallel database solutions. This is, however, expensive and hard to maintain and is against the pay-as-you-use model of the cloud. All this points to using a scalable cloud service such as SimpleDB to store provenance, as we do in two of our protocols (Section 4.3.2 and Section 4.3.3). Storing the provenance in a separate service opens the issue of coordinating updates between the database service and object store service, which we address while describing the protocols.

Some of the properties of the cloud, on the other hand, make storing provenance easier. For example, NFS and the file system have to ensure consistency in the face of partial object writes, while cloud stores deal only with complete objects. Hence cloud provenance does not have to consider partial write failures.

4.2 Architecture Overview

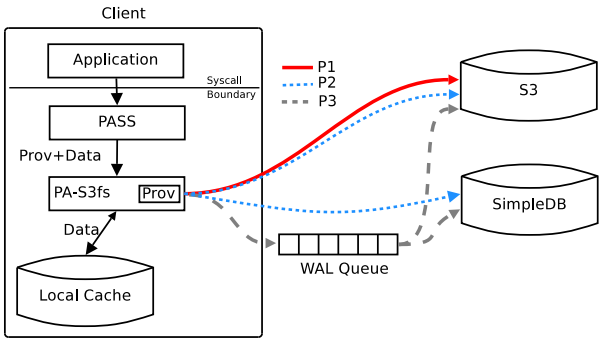


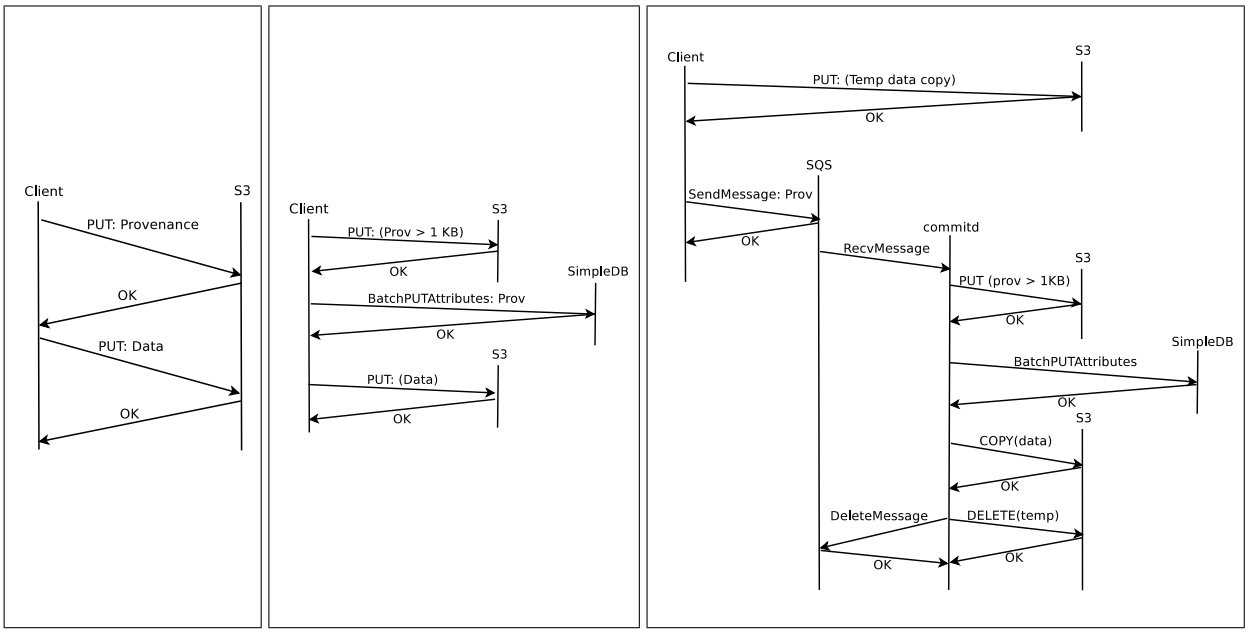
Figure 1: Architecture: The figure shows how provenance is collected and the cloud is used as a backend.

Figure 1 shows our system architecture. The system is composed of the client (compute node) and the cloud. The client is in turn composed of PASS and PA-S3fs. PASS monitors system calls, generating provenance and sending both provenance and data to Provenance Aware S3fs (PA-S3fs). PA-S3fs, a user-level provenance-aware file system interface for Amazon's S3 storage service, caches data and provenance on the client to reduce traffic to S3. PA-S3fs caches data in a local temporary directory and the provenance in memory. On certain events, such as file close or flush, it sends both the data and the provenance to the cloud using one of the protocols P1, P2, or P3, which we discuss in the next subsections. Further, PASS has algorithms built into it that preserve causality by carefully creating logical versions of objects when they are simultaneously updated by multiple processes at the same client [29]. The provenance recorded in the cloud by the protocols reflects this versioning.

Implementation PA-S3fs is derived from S3fs [36], a user-level FUSE [19] file system that provides a file system interface to S3. PA-S3fs extends S3fs by interfacing it to PASS, our collection infrastructure. PASS internally uses the Disclosed Provenance API (DPAPI) [28] to satisfy the properties specified in Section 3 and eventually stores the provenance on a backend that exports the DPAPI. Hence, extending S3fs to PA-S3fs translates to extending S3fs and FUSE to export the DPAPI.

4.3 Protocols

Table 1 summarizes our three protocols with respect to the properties in Section 3. Although we discuss the protocols in the context of moving data from users to the cloud, they can also be used while replicating data and provenance across different cloud service providers. Further, while our implementation is based on extending the file system interface to the cloud, the protocols are



(a) (b) (c)

Figure 2: Protocol 1 (a): Both provenance and data are recorded in a cloud object store (S3). Protocol 2 (b): Provenance is stored in a cloud database (SimpleDB) and data is stored in a cloud store (S3). Protocol 3 (c): Provenance is stored in a cloud database (SimpleDB) and data is stored in a cloud store (S3). A cloud messaging service (SQS) is used to provide data-coupling and multi-object causal ordering.

Property	P1	P2	P3
Provenance Data-Coupling	✖	✖	✔
Multi-object Causal Ordering	✔	✔	✔
Efficient Query	✖	✔	✔

Table 1: Properties Comparison. A check mark indicates that the property is supported, otherwise it is not.

independent of the storage model and applicable whenever provenance has to be stored on the cloud.

4.3.1 P1: Standalone Cloud Store

Storage Scheme: We map each file to an S3 object and store the object's provenance as a separate S3 object. It might seem attractive to record provenance as metadata of the object, but that introduces two problems. First, removing the object removes its provenance, violating provenance persistence. Second, most systems impose a hard limit on the size of an object's metadata. To address the deletion issue, one could truncate the data in the object and rename the object to a shadow directory on deletion. To address the metadata limit, one could store the extra provenance in the first n bytes of the object itself and on deletion, truncate the data part of the object. Instead, we create a primary S3 object containing the data and a second, provenance S3 object, named with a `uuid` and containing the primary object's provenance plus an additional provenance record containing the name of the primary S3 object. In the primary S3 object's metadata, we record a version number and the `uuid`, thus linking the data and its provenance. For objects that are not persistent, such as pipes and processes, we record only the provenance object with no primary object. For provenance queries, this scheme requires us to lookup the primary object and then retrieve the provenance whereas the previous scheme can avoid this. On deletions, however, the previous scheme requires the system to update all provenance referring to the object to point to the new name assigned on deletion. We chose to store provenance in a separate object, because provenance queries are infrequent relative to object operations, and updating provenance pointers on every delete can be expensive.

Protocol: Figure 2a depicts protocol P1. On a file close (or flush), we perform the following operations:

1. Extract the provenance of the file (cached by PA-S3fs). PUT the provenance into the S3 provenance object. If the provenance object already exists, GET the existing object, append the new provenance to it, and then issue a PUT.

2. PUT the data object with metadata attributes containing the name of the provenance object and the current version.

Before sending the provenance and data of an object, we need to identify the ancestors of the object and send any unrecorded ancestors and their provenance to ensure multi-object causal ordering. A client can, at best, assure a consistency model comparable to that of the underlying system; that is if the underlying system supports eventual consistency, then the best P1 can do is ensure *eventual multi-object causal ordering*. A reading client that wants to check multi-object causal ordering must use Merkle hash trees or some similar scheme to verify the property. If the property is not satisfied, the client should try refreshing the data until the objects do meet the multi-object causal ordering property.

Discussion: This protocol does not support data-coupling, but using version numbers stored both in the provenance object and the primary object's metadata, clients can detect provenance decoupled from data. P1 achieves eventual multi-object causal ordering if it sends all the ancestors of an object and their provenance to S3 before sending the object's provenance to S3. However, such an implementation can suffer from high latency. Querying is inefficient as we cannot retrieve objects by their individual provenance attributes; we can only retrieve all of an object's provenance via a GET call. If we do not know the exact object whose provenance we seek, then we need to iterate over the provenance of every object in the repository, which is so inefficient as to be impractical.

4.3.2 P2: Cloud Store with a Cloud Database

Storage Scheme: This scheme, which is already independently in use by some cloud users [13], stores each file as an S3 object and the corresponding provenance in SimpleDB. We store the provenance of a version of an object as one SimpleDB item (row in traditional databases). As in P1, we reference the provenance of an object by `uuid` assigned to the object at creation time. For example, assume that an object named *foo* has `uuid` 'uuid1', its version is 2, and it has two provenance records: (input, bar_2) and (type, file). P2 stores this in SimpleDB as:

```
ItemName=uuid1_2
attribute-name=name,attribute-value=foo
attribute-name=input,attribute-value=bar_2
attribute-name=type,attribute-value=file
```

The name attribute allows us to find an object from its provenance. We chose this one-row-per-version scheme instead of storing the provenance of all versions of an object as one SimpleDB item, as it allows users to distinguish the version to which the provenance belongs. We store provenance values larger than the 1KB SimpleDB limit as separate S3 objects, referenced from items in SimpleDB. As in P1, we store the object's current version number and uuid in its metadata.

Protocol: Figure 2b shows the second protocol. On a file close, we extract the provenance cached in memory and convert it to attribute-value pairs. We then group the attribute-value pairs by file version, construct one item for the provenance of each version of the file, and perform the following actions:

1. If any of the values are larger than 1KB, store them as S3 objects and update the attribute-value pair to contain a pointer to that object.
2. Store the provenance in SimpleDB by issuing *BatchPutAttributes* calls. SimpleDB allows us batch up to 25 items per call, hence we issue as many calls as necessary to store all the items.
3. PUT the data object with metadata attributes containing the name of the provenance object and the current version.

As in P1, P2 enforces multi-object causal ordering by recording ancestors and their provenance before sending the provenance and data of the new object.

Discussion: P2 is an improvement over P1 in that it provides efficient provenance queries, because we can retrieve indexed provenance from SimpleDB. Like P1, P2 does not provide data-coupling but can detect coupling violations and exhibits high latency to ensure multi-object causal ordering. Due to eventual consistency, we can encounter a scenario in which SimpleDB returns old versions of provenance when S3 returns more recent data (and vice versa). We detect this by comparing the version of the object in S3 and the version returned in the provenance. If they are not consistent, we can request the specific version of the provenance we need from SimpleDB.

4.3.3 P3: Cloud store with Cloud Database and Messaging Service

Storage Scheme and Overview: P3 uses the same S3/SimpleDB storage scheme as P2, but differs from P2 in its use of a cloud messaging service (SQS) and transactions to ensure provenance data-coupling. Each client has an SQS queue that it uses as a write-ahead log (WAL) and a separate daemon, the *commit daemon*, that reads the log records and assembles all the records belonging to a transaction. Once it has all the records for a transaction, the daemon pushes data in the records to S3 and provenance to SimpleDB. If the client crashes before it can log all the packets of a transaction to the WAL queue, the commit daemon ignores these records. One might be tempted to use a local log instead of an SQS

queue, but such an arrangement leads to data-coupling violations when a client crashes before the commit daemon has completely committed a transaction. By using SQS as the log, if the client running the commit daemon crashes during a commit, another machine can commit the partially completed transaction.

Messages on SQS (and Azure) cannot exceed 8KB, hence we cannot directly record large data items in the WAL queue. Instead, we store large objects as temporary S3 objects, recording a pointer to the temporary object in the WAL queue. The commit daemon, while processing the WAL queue entries, copies a temporary object to its real object and then deletes the temporary object. Both S3 and Azure do not currently support a rename operation. Hence the object has to be copied from the temporary name to the real object. One thousand copy operations cost 0.01 USD for S3 and 0.001 USD for Azure with no charge for the data transfer required to perform the copy. Hence the copy operation has minimal cost from a user's perspective. Once items are in the WAL queue, they are guaranteed to eventually be stored in S3 or SimpleDB, so the order in which we process the records does not matter.

We must, however, *garbage collect* state left over by uncommitted transactions. SQS automatically deletes messages older than four days, so we do not need to perform any additional reclamation (unless the 4-day window becomes too large) on the queue. However, temporary objects that have been stored on S3 must be explicitly removed if they belong to uncommitted transactions. We use a *cleaner* daemon to remove temporary objects that have not been accessed for 4 days.

Protocol: Figure 2c shows our final protocol. We divide the protocol into two phases: log and commit. The log phase begins when an application issues a `close` or flush on a file and consists of the following actions.

1. Store a copy of the data file with a temporary name on S3.
2. Allocate a uuid as a transaction id. Extract the provenance of the object. Group the provenance records into chunks of 8KB and store each of these chunks as log records (messages) in the WAL queue. The first bytes of each message contain the transaction id and a packet sequence number. The first message has the following additional records: A record indicating the total number of packets in the transaction, a record that has a pointer to the temporary object, and a record tagged with the transaction id and the object version.

In the commit phase, the commit daemon assembles the packets belonging to transactions and once it receives all the packets of a transaction, performs the following actions.

1. Store any provenance record larger than 1KB into a separate S3 object and update the attribute-value pair to contain a pointer to the S3 object.
2. Store the provenance in SimpleDB by issuing *BatchPutAttributes* calls. SimpleDB allows us batch up to 25 items per call, hence we issue as many calls as necessary to store all the items.
3. Execute an S3 *COPY* method to copy the temporary S3 object to its permanent S3 object, updating the version as part of the COPY.
4. Delete the temporary S3 object using the S3 *DELETE* method. Delete all the messages related to the transaction from the WAL queue using the SQS *DeleteMessage* command.

We include all not-yet-written ancestors of an object in the object's transaction in order to obtain multi-object causal ordering. This ensures that we maintain multi-object causal ordering even if we send packets in parallel to SQS. In contrast, the previous protocols required that we carefully order ancestors and their descendants.

Discussion: The protocol satisfies eventual provenance data-coupling. We cannot provide a stronger guarantee due to the eventual consistency model of the services and due to the fact that we cannot modify the underlying services. Applications that are sensitive to provenance data-coupling can detect inconsistency and can retry again on detecting inconsistency. In prior work, we discuss provenance-aware `read` and `write` system calls [28], which provide an interface that can perform these checks on behalf of the application. Similar to the previous protocols, this protocol maintains eventual multi-object causal ordering, but provides better throughput. Further, queries are executed efficiently as SimpleDB provides rapid, indexed lookup.

5 Evaluation

The goal of our evaluation is to understand the relative merits of the different protocols and their feasibility in practice. To that end, our evaluation has three parts: first, we quantify the storage utilization and data transfer of the protocols independent of the provenance collection framework (Section 5.1), second, we evaluate the efficacy, performance, and cost of the protocols under various workloads (Section 5.2), and third, we evaluate the query performance of the protocols (Section 5.3).

We used the following software configurations for the evaluation:

- **S3fs**: S3fs on a vanilla Linux 2.6.23.17 kernel.
- **P1**: Provenance-Aware S3fs on a PASS kernel (Linux 2.6.23.17 kernel with appropriate modifications), with both provenance and data being recorded on S3.
- **P2**: Provenance-Aware S3fs on a PASS kernel with provenance stored on SimpleDB.

- **P3**: Provenance-Aware S3fs on a PASS kernel with provenance on SimpleDB, with an SQS queue used as a log.

To maximize performance, we implemented the protocols to upload the data objects, their provenance, and ancestral data and provenance in parallel (this violates multi-object causal ordering for P1 and P2).

We used Amazon *EC2 Medium* [15] instances running Fedora 8 to run the benchmarks. The medium instance configuration at the time we ran the experiments was a 32-bit platform with 1.7 GB of memory, 5 EC2 Compute Units (2 virtual cores with 2.5 EC2 Compute Units each), and 350 GB of instance storage. Since one cannot install a custom kernel on EC2 instances, we run the workload benchmarks (Section 5.2) that use the vanilla Linux kernel and the PASS kernel as User Mode Linux (UML) [14] instances with 512MB of RAM on EC2 machines. We had to use medium EC2 machines as the small instances proved to be insufficient to run the PASS kernel as a UML instance. We also ran the benchmarks from one of our local machines. Both the usage models, i.e, running the workloads on local machine and storing data and provenance on the cloud or running the workloads on EC2 machines and storing the data and provenance on the cloud are valid as our protocols are agnostic to the usage model.

We used the following three workloads in our evaluation. Each of the three workloads represents provenance trees of different depths.

CVSROOT nightly backup This workload simulates nightly backups of a CVS repository by extracting nightly snapshots from 30 days of our own repository, creating a `tarball` for each night, and uploading the 30 snapshots to AWS. The provenance tree for this workload is nearly flat with just the program `cp` as the ancestor of the stored archives. The workload is IO intensive, has negligible compute time, and S3fs performs 240 operations under this workload.

Blast This is a biological workload representative of scientific computing workloads. Blast is a tool used to find protein sequences that are closely related in two different species. This workload simulates the typical Blast job observed at NIH [12]. The provenance tree of the workload has a depth of five. The workload has a mix of compute and IO operations and S3fs performs 10,773 operations under this workload.

Challenge This is the workload used in the first and second provenance challenge [35]. The workload simulates an experiment in fMRI imaging. The inputs to the workload are a set of new brain images and a single reference brain image. First, the workload normalizes the images with respect to the reference image. Second, it transforms the image into a new image. Third, it averages all the transformed images into one single image. Fourth, it slices the average image in each of three dimensions to produce a two-dimensional atlas along a plane in the third dimension. Last, it converts the atlas data set into a graphical atlas image. The challenge workload graph is the deepest with maximum path length of eleven. Similar to blast, the workload has a mix of compute and IO operations and S3fs performs 6,179 operations.

We ran each workload at least 5 times for each configuration. The elapsed times we present do not include the commit daemon times for P3 as it operates asynchronously, thus not affecting the elapsed times.

Our evaluation results are AWS-specific as it is currently the only mature cloud service that also provides all the services we need (Note that SimpleDB, as of January 2010, is in public beta). Further, we find that AWS performance is highly variable due to a variety of factors that are not under our control, such as the load on the services, WAN network latencies, and the version of the software used for the service. Further, upgrades to the services seem to continually improve performance over time, thus making reproducibility harder. Due to the variance, we find that results from different days are not comparable. We found that we needed to execute the benchmarks at the same time or within a short time period for the results to be comparable. Even so, we find that at a given time, any of the protocols can perform well due factors such as relative load on the service, proximity of the replica chosen to service requests, etc. We have run a large number of experiments between August 2009 and January 2010. The results we present are those that are most representative of the behavior we observed and best illustrate the trends that we observed repeatedly.

5.1 Microbenchmarks

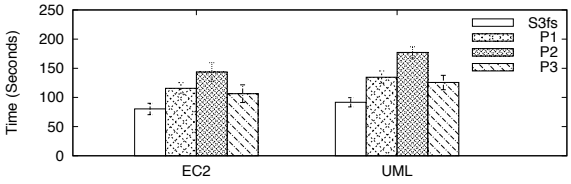


Figure 3: Elapsed times for the microbenchmark on an EC2 instance and on an UML machine running on an EC2 instance.

Our microbenchmarks quantify the throughput obtained by each protocol relative to S3fs. To isolate the protocol throughput from the application and provenance collection overheads, we ran the Blast benchmark

on a unmodified PASS system and captured the provenance. We then built a tool that uploaded the data objects and their provenance to the cloud using each protocol. We ran the microbenchmark on an EC2 instance. Further, to demonstrate that the results in the following section are not an artifact of using UML, we also ran the microbenchmark on a UML instance running on EC2. Figure 3 shows the microbenchmark results.

On EC2, P3, the protocol that best satisfies our properties, also exhibits the lowest overhead (32.6%) and P1 dominates P2. As there is no application time in this microbenchmark, the overheads are relatively high for all the protocols, ranging from 32% for P3 to 78.9% for P2. The UML microbenchmark results follow the pattern we see in the EC2 microbenchmark results, indicating that UML does not change the relative performance of the protocols.

	S3	**SimpleDB**	**SQS**
Time (s)	324.7	537.1	36.2

Table 2: Time taken to upload 50MB of provenance to each of the services.

To understand why the protocols exhibit this relative performance, we ran another benchmark where we uploaded, in parallel, the first 50MB of provenance generated during a Linux compile to each of S3, SimpleDB, and SQS. Table 2 shows the results of this experiment. We find that SQS is dramatically faster than either S3 or SimpleDB and that S3 is significantly faster than SimpleDB. We tried to find the maximum possible throughput by varying the number of concurrent connections to each service. We found that S3 and SQS scaled well as the number of connections increased (we stopped at 150) while SimpleDB peaked at around 40 concurrent connections from a single client host. The numbers in Table 2 used 150 concurrent connections for S3 and SQS and 40 concurrent connections for SimpleDB. Thus, P1 leverages the better parallelism in S3 relative to SimpleDB and outperforms P2. P3 exhibits the best performance as it bundles all its provenance into 8KB chunks uploading them to SQS, the fastest service.

Table 3 shows the data and operation overheads. The data overheads are negligible – all under 1%. In contrast, the overhead in terms of number of operations is quite large, because all the protocols are at least doubling their work, writing both provenance and data. But, as we will see in the next section, operations are not very expensive.

5.2 Workload Overheads

Figure 4 shows the elapsed times for the workload benchmarks run from EC2 instances and from a local machine. We present results collected during September 2009 (Figure 4a) and during December and January 2009-2010 (4b). The Figure consists of 12 sets of results, with each set consisting of 3 individual results that measure the individual protocol overhead relative to S3fs.

	Data Transmitted (MB)	**Operations**
S3fs	713.09	617
P1	715.31 (0.31%)	2287 (270.7%)
P2	716.11 (0.42%)	1235 (100.2%)
P3	716.32 (0.45%)	1337 (116.7%)

Table 3: Data transfer and operation overheads for the protocols. The overheads, shown in parentheses, are relative to S3fs. Protocol P3 numbers do not include the commit daemon. The operation count in the microbenchmark are reduced as we only upload the final results of the computation.

Overall, we observe that the overheads are reasonable – less than 10% for 29 of the 36 individual results shown above. Of the remaining 7 results, 5 of them have an overhead less than 20%. The maximum overhead is 36% for P2 for the challenge workload benchmark run in December/January on EC2. For the same scenario in September, P2 has an overhead of 24.3%.

Incorporating application time into the equation reveals that the relative performance of the different protocols is comparable. At first blush, P3 seems to be the fastest protocol as it performs the best in 8 out of the 12 result sets. However, the error bars on the graphs indicate that the difference is not statistically significant.

We expected the elapsed time for the benchmarks to be greater in the local machine case than in the EC2 case. This was borne out for the nightly backup and challenge workloads. However, the Blast workload ran faster on the local machine than on EC2. We hypothesized that this was caused by an interaction between Blast's memory accesses and the UML's small 512MB memory (512MB is the maximum UML instance memory). We confirmed this by running Blast and the nightly backup benchmark on a native (not UML) EC2 instance. The I/O time for the nightly benchmark increased from 419s on a raw EC2 machine to 528s on a UML EC2 instance. For Blast, the corresponding number increases from 650s to 1322s. The dramatic difference between native EC2 and UML EC2 for the Blast workload was highly suggestive.

Finally, we observe that the elapsed times for all benchmarks except for the nightly local case, have reduced between 4% to 44.5% from September 09 to December 09/January 10. We also observe that P1's performance approaches that of P3 in many of the application benchmarks. As we stated earlier, this is due to various factors that are beyond our control.

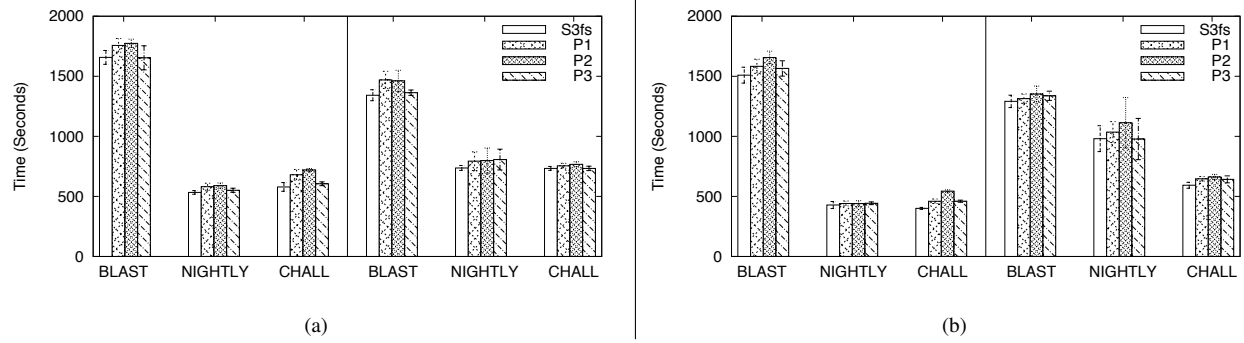


Figure 4: Elapsed times for workload benchmarks. Figure 4a shows the results for the benchmarks from September 2009. Figure 4b shows the results for the benchmarks from December 2009/January 2010. In both graphs, the left half shows elapsed times when the benchmark runs on EC2 instances. The right half shows the elapsed time when running on a local machine.

	Nightly	**Blast**	**Challenge**
S3fs	$1.05	$0.37	$0.27
P1	$1.05	$0.39	$0.29
P2	$1.05	$0.38	$0.29
P3	$1.06	$0.40	$0.30

Table 4: Cost for each benchmark (includes commit daemon cost).

Table 4 shows the cost in USD for each protocol. Overall, we observe the following relationship between protocols: $P3 > P1 >= P2 >= S3fs$. The extra cost required to store provenance in each of the protocols is minimal (compared to S3fs). As expected, P3 is the most expensive due to the operations it performs to log provenance on SQS and then upload provenance to SimpleDB. The cost for P1 and P2 are similar for Nightly and Challenge workloads. For Blast, P2 is cheaper than P1, because P1 needed more operations to store the provenance on S3 than P2 required to store the same provenance on SimpleDB.

5.3 Query performance

To evaluate query performance, we ran the following four queries on the Blast workload provenance:

Q.1 Retrieve all the provenance ever recorded.
Q.2 Given an object, retrieve the provenance of all versions of the object.
Q.3 Find all the files that were directly output by *Blast*.
Q.4 Find all the descendants of files derived from *Blast*.

We chose these queries as they represent varying levels of complexity. The first query is a simple dump of all the provenance. The second query uses an object handle to retrieve all of its provenance but requires no search. The third involves a lookup and a single-level descendant query. The fourth is a full descendant query. Table 5 shows the query results. There are only two different sets of results as P1 uses S3 objects to store provenance, and P2 and P3 use SimpleDB to store provenance, thus having identical query capabilities and performance.

We implement Q.1 in S3 by fetching the list of all S3 provenance objects and then performing a GET for each. Since there are no ordering constraints on when the GET requests are executed, i.e., it is not necessary for any GET to wait for the completion of another request, parallelizing these operations greatly improves performance (as we can see in the Table 5).

In SimpleDB, we execute "SELECT *" to retrieve all the provenance. We implement this as a single request that, due to the limits imposed by SimpleDB, has to be decomposed into several sequential operations, where one operation has to complete before the next one can start, so this request cannot be parallelized. However, the number of SimpleDB round-trips is smaller than in S3, and the query thus executes much more quickly.

In Q.2, the performance is comparable for both S3 and SimpleDB. We implement this query by first issuing a HEAD operation on the object to determine the uuid used to reference its provenance. In S3, we then issue a GET on the provenance object, while in SimpleDB we perform an appropriate SELECT operation. Note that these two operations must be performed sequentially, so the query cannot benefit from parallelism. Because both S3 and SimpleDB perform the HEAD operation, the performance is comparable.

In Q.3 and Q.4, we need to first find records (items) of processes that correspond to the multiple executions of *Blast*. This translates into looking up all items that satisfy a certain property. In S3, this requires a scan of all provenance objects. We implemented these two queries in S3 by retrieving all provenance objects and then processing the query locally. SimpleDB is more ef-

Query	S3 (P1)				SimpleDB (P2, P3)			
	Time (s)		MB Transferred	Ops.	Time (s)		MB Transferred	Ops.
	Sequential	Parallel			Sequential	Parallel		
Q.1	48.57	7.04	2.95	1671	0.83	–	2.05	13
Q.2	0.060	–	0.0015	2	0.037	–	0.008	2
Q.3	48.57	7.04	2.95	1671	0.82	0.34	0.11	37
Q.4	48.57	7.04	2.95	1671	1.86	0.72	0.19	87

Table 5: Query performance. The table shows the time taken to complete the queries, the total data transferred, and the total number of executed operations. The table shows the times for both sequential and parallel execution of the query. In both cases, the number of operations and the data transferred was the same. For Q.2, the values shown are the average time taken per object.

ficient for Q.3 and Q.4 as it indexes all the attributes in the database. Hence, for Q.3 and Q.4 in SimpleDB, we first issue a SELECT to find all items corresponding to *Blast*. We then issue a set of SELECT queries to find the names of all the items that reference the *Blast* items retrieved in the previous call. For Q.4, we have to repeat the second step recursively until we have located all the descendants. As we can see from the results, SimpleDB is an order of magnitude faster as it can retrieve data more selectively. Further, the performance gap between S3 and SimpleDB is bound to grow larger as more objects are involved.

5.4 Summary

All three protocols have low cost and data transfer overheads. The workload overheads were less than 10% over S3fs for all protocols in the majority of the cases. Our microbenchmarks show that P3, our most robust protocol, is the best performing. But, when application overheads are included, all protocols are within statistical error. Thus users can select the best protocol best suited for their needs, without performance penalty.

6 Related Work

Provenance in distributed workflow-based and grid environments has been explored by several prior research projects [11, 17, 21, 40]. There are also systems that track application-specific data to be able to regenerate data [23] or reproduce experiments [16]. All prior work assumes the ability to alter the underlying system components, as opposed to having to make due with a given infrastructure as we do here. We develop a provenance solution atop an infrastructure over which we have no control. However, we complement this prior work, and our protocols can be used to move the provenance collected by the above frameworks to the cloud.

Branthner et. al. [9] explore using S3 as a backend for a database. They use SQS as a log to ensure atomic updates to the database, similar to the mechanism we use in P3. While the mechanisms are similar, this work and Branthner et. al. address different research questions. Brantner et. al. use the mechanism to coordinate updates to a single service. We use the mechanism to provide consistency between two services, S3 and SimpleDB.

In prior work [31], we explored the challenges of storing provenance in the cloud, outlined protocols, and performed a rudimentary analysis of the protocols. This work follows on where that work left off, i.e., we implement and evaluate the protocols. Some tweaks were necessary to realize the protocols in practice. For example, for P1, we had originally intended to store the provenance as metadata of the S3 object, but this does not satisfy the data independent persistence property.

Hasan et. al. [22] discuss cryptographic mechanisms to protect provenance from tampering. Juels et. al [24] and Ateniese et. al. [4] present schemes that allow users to efficiently verify that a provider can produce a stored file. These research projects are complementary to our work and we can leverage them to verify that malicious users and servers have not tampered provenance on the cloud.

7 Native Cloud Provenance: Research Challenges

This work has focused on storing and accessing provenance on current cloud offerings. In the current scheme where provenance and data are stored on separate services, however, providers have no means to link the provenance of an object to its data. Providing native support for provenance on cloud stores enables providers to relate provenance to its data, allowing the providers to leverage the provenance for their benefit [32]. For example, the graph structure in provenance can provide service providers with hints for object replication. As more data moves to the cloud, providers will need to provide search capabilities to users. As outlined previously (Section 2.2), provenance can play a crucial role in improving search quality. Cloud providers could also allow users to chose between storing data and regenerating data on demand, if the provenance of data were available to them [3].

Building native support for the cloud presents a number of challenges in addition to the issues that arise in

building large scale distributed systems. We discuss some of these research challenges next.

System Architecture A native provenance store has to support both the object storage requirements of data and the database functionality requirements of provenance. The simplest approach is to obviously store the provenance and the data in two separate services. However, one needs to to co-ordinate updates across the two services. To provide strong provenance data-coupling using an external co-ordination service, the underlying services have to export a transactional interface. However, a fully transactional system is not feasible at the scales at which the cloud operates. Finding a middleground between the two extremes and the cost of each approach (the naive approach, fully transactional, and a possible middleground) is an open research challenge.

Security Provenance can potentially contain sensitive information. The fundamental issue is that provenance and the data it describes do not necessarily share the same access control. For example, consider a report generated by aggregating the health information of patients suffering a certain ailment. While the report (the data) can be accessible to the public, the files that were used to generate the report (the provenance) must not be. Provenance security is an open problem that is being explored by multiple research groups [10]. Providers need to take these issues into consideration while extending their service to support provenance.

Provenance Storage The semi-structured data model, imported by SimpleDB and Azure Table, is appropriate for storing provenance graphs. These services, however, are not necessarily optimized to store provenance graphs. Recently, databases such as Neo4j [33], have been designed from the ground-up for storing graphs. Exploring if a data service designed from the ground-up for storing provenance is more efficient in terms of performance and cost compared to a generic database service is an interesting avenue for future work.

Learning Models As we stated above, cloud providers can take advantage of provenance in a variety of ways. However, for each particular application, a particular subset of provenance has to be extracted or a particular type of generalization has to be made across all objects. For some applications, a simple pattern matching approach might be sufficient and for other applications, sophisticated machine learning mechanisms might be necessary. The models necessary to extract the necessary data for each application is an open question.

Processing Provenance Graphs The models above need to process the provenance graph to extract the necessary information. However, there are currently no general purpose graph processing systems available. MapReduce is one mechanism that is generally used to process graphs. Pregel [26], based on Bulk Synchronous Parallel model, is another approach that is currently being developed. How the two mechanisms compare with each other for graph workloads is a study worth undertaking.

Transparent Provenance Collection This work expects and trusts users to supply provenance. The provenance graph supplied by users is rich as it consists of process information. Without support from users, the cloud can automatically infer diluted provenance, i.e., provenance minus process information. In this provenance graph, all the processes from a single host will be represented by a single node representing the host. What subset of the provenance applications can be driven by this diluted graph?

Economics Providing native support for provenance increases the cost to the provider in terms of storage, CPU, and network bandwidth. Prior to embarking on building a native cloud store, an economic analysis that justifies that the extra cost of provenance is necessary. To this end, we need to design appropriate economic models and evaluate the cost of storing provenance.

8 Conclusions

The cloud is poised to become the next generation computing environment, and we have shown that we can add provenance to cloud storage in several ways. Our evaluation shows that all three protocols have reasonable overhead in terms of time to execute and minimal financial overhead. Further, our most robust protocol, which provides all the properties we outline, performs as well, if not better, than the other protocols, making it one of those rare occasions where we need not make compromises to achieve our objectives. We can construct a fully functional and performant provenance system for the cloud using off-the shelf cloud components.

The web, which is the most widely used medium for sharing data, does not provide data provenance. The cloud, however, is still in its infancy, and can easily incorporate provenance now. We can deploy these kinds of services with systems today, but it is worth investigating the cost, efficacy, and feasibility of offering provenance as a native cloud service as well.

Acknowledgments We thank Kim Keeton, Bill Bolosky, Keith Smith, Erez Zadok, James Hamilton, and Nick Murphy for their feedback on early drafts of the paper. We thank Matt Welsh for discussions at early stages of the project. We thank Jason Flinn, our shepherd, for repeated careful and thoughtful reviews of our paper. We thank Kurt Messersmith from Amazon Web Services for providing us with credits to run the experiments in the paper. We thank the FAST reviewers for the valuable feedback they provided. This work was partially made possible thanks to NSF grant CNS-0614784.

References

[1] Pubchem. http://pubchem.ncbi.nlm.nih.gov/.

[2] Genbank. *Nucleic Acids Research 36 (Database Issue)* (January 2008).

[3] ADAMS, I., LONG, D. D. E., MILLER, E. L., PASUPATHY, S., AND STORER, M. W. Maximizing efficiency by trading storage for computation.

[4] ATENIESE, G., BURNS, R., CURTMOLA, R., HERRING, J., KISSNER, L., PETERSON, Z., AND SONG, D. Provable data possession at untrusted stores. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security* (New York, NY, USA, 2007), ACM, pp. 598–609.

[5] Amazon Web Services. http://aws.amazon.com.

[6] Windows Azure Blob. http://go.microsoft.com/fwlink/?LinkId=153400.

[7] Windows Azure Queue. http://go.microsoft.com/fwlink/?LinkId=153402.

[8] Windows Azure Table. http://go.microsoft.com/fwlink/?LinkId=153401.

[9] BRANTNER, M., FLORESCU, D., GRAF, D., KOSSMANN, D., AND KRASKA, T. Building a database on S3. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2008), ACM, pp. 251–264.

[10] BRAUN, U., SHINNAR, A., AND SELTZER, M. Securing Provenance. In *Proceedings of HotSec 2008* (July 2008).

[11] CHEN, Z., AND MOREAU, L. Implementation and evaluation of a protocol for recording process documentation in the presence of failures. In *Proceedings of Second International Provenance and Annotation Workshop (IPAW'08)*.

[12] COULOURIS, G. Blast benchmarks. http://fiehnlab.ucdavis.edu/staff/kind/Collector/Benchmark/Blast_Benchm%ark.

[13] DAGDIGIAN, C. Plenery Keynote: Bio.IT World. http://blog.bioteam.net/wp-content/uploads/2009/04/bioitworld-2009-keyn%ote-cdagdigian.pdf.

[14] DIKE, J. User-mode linux. In *Proceedings of the 5th Annual Linux Showcase & Conference* (Oakland, California, USA, 2001).

[15] Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2.

[16] EIDE, E., STOLLER, L., AND LEPREAU, J. An experimentation workbench for replayable networking research. In *4th USENIX Symposium on Networked Systems Design & Implementation* (2007).

[17] FOSTER, I., VOECKLER, J., WILDE, M., AND ZHAO, Y. The Virtual Data Grid: A New Model and Architecture for Data-Intensive Collaboration. In *CIDR* (Asilomar, CA, Jan. 2003).

[18] FREW, J., METZGER, D., AND SLAUGHTER, P. Automatic capture and reconstruction of computational provenance. *Concurrency and Computation: Practice and Experience 20* (April 2008), 485–496.

[19] Filesystem in userspace. http://fuse.sourceforge.net/.

[20] GRAY, J., SLUTZ, D., SZALAY, A., THAKAR, A., VANDENBERG, J., KUNSZT, P., AND STOUGHTON, C. Data Mining the SDSS SkyServer Database. Research Report MSR-TR-2002-01, Microsoft Research, January 2002.

[21] GROTH, P., MOREAU, L., AND LUCK, M. Formalising a protocol for recording provenance in grids. In *Proceedings of the UK OST e-Science Third All Hands Meeting 2004 (AHM'04)* (Nottingham, UK, Sept. 2004). Accepted for publication.

[22] HASAN, R., SION, R., AND WINSLETT, M. The Case of the Fake Picasso: Preventing History Forgery with Secure Provenance. In *FAST* (2009).

[23] HEYDON, A., LEVIN, R., MANN, T., AND YU, Y. *Software Configuration Management Using Vesta*. Monographs in Computer Science, Springer, 2006.

[24] JUELS, A., AND KALISKI, JR., B. S. Pors: proofs of retrievability for large files. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security* (New York, NY, USA, 2007), ACM, pp. 584–597.

[25] The LINQ project. http://msdn.microsoft.com/en-us/vcsharp/aa904594.aspx.

[26] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: a system for large-scale graph processing. In *PODC '09: Proceedings of the 28th ACM symposium on Principles of distributed computing* (New York, NY, USA, 2009), ACM, pp. 6–6.

[27] MARGO, D. W., AND SELTZER, M. The case for browser provenance. In *1st Workshop on the Theory and Practice of Provenance* (2009).

[28] MUNISWAMY-REDDY, K.-K., BRAUN, U., HOLLAND, D. A., MACKO, P., MACLEAN, D., MARGO, D., SELTZER, M., AND SMOGOR, R. Layering in provenance systems. In *Proceedings of the 2009 USENIX Annual Technical Conference*.

[29] MUNISWAMY-REDDY, K.-K., AND HOLLAND, D. A. Causality-Based Versioning. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies* (Feb 2009).

[30] MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., BRAUN, U., AND SELTZER, M. Provenance-aware storage systems. In *Proceedings of the 2006 USENIX Annual Technical Conference*.

[31] MUNISWAMY-REDDY, K.-K., MACKO, P., AND SELTZER, M. Making a cloud provenance-aware. In *1st Workshop on the Theory and Practice of Provenance* (2009).

[32] MUNISWAMY-REDDY, K.-K., AND SELTZER, M. Provenance as first-class cloud data. In *3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS'09)* (2009).

[33] Neo4j, the graph database. http://neo4j.org/.

[34] Provenance aware service oriented architecture. http://twiki.pasoa.ecs.soton.ac.uk/bin/view/PASOA/WebHome.

[35] The First Provenance Challenge. http://twiki.ipaw.info/bin/view/Challenge/FirstProvenanceChallenge.

[36] RIZUN, R. S3fs: FUSE-based file system backed by Amazon S3. http://code.google.com/p/s3fs/wiki/FuseOverAmazon.

[37] Amazon Simple Storage Service (Amazon S3). http://aws.amazon.com/s3.

[38] Amazon SimpleDB. http://aws.amazon.com/simpledb.

[39] SHAH, S., SOULES, C. A. N., GANGER, G. R., AND NOBLE, B. D. Using provenance to aid in personal file search. In *Proceedings of the USENIX Annual Technical Conference* (2007).

[40] SIMMHAN, Y. L., PLALE, B., AND GANNON, D. A framework for collecting provenance in data-centric scientific workflows. In *ICWS '06: Proceedings of the IEEE International Conference on Web Services* (2006).

[41] Amazon Simple Queue Service (SQS). http://aws.amazon.com/sqs.

[42] ZHAO, J., GOBLE, C.AND GREENWOOD, M., WROE, C., AND STEVENS, R. Annotating, linking and browsing provenance logs for e-science.

I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance

Ricardo Koller — rkoll001@cs.fiu.edu
Raju Rangaswami — raju@cs.fiu.edu
School of Computing and Information Sciences, Florida International University

Abstract

Duplication of data in storage systems is becoming increasingly common. We introduce I/O Deduplication, a storage optimization that utilizes content similarity for improving I/O performance by eliminating I/O operations and reducing the mechanical delays during I/O operations. I/O Deduplication consists of three main techniques: *content-based caching*, *dynamic replica retrieval*, and *selective duplication*. Each of these techniques is motivated by our observations with I/O workload traces obtained from actively-used production storage systems, all of which revealed surprisingly high levels of content similarity for both stored and accessed data. Evaluation of a prototype implementation using these workloads revealed an overall improvement in disk I/O performance of 28-47% across these workloads. Further breakdown also showed that each of the three techniques contributed significantly to the overall performance improvement.

1 Introduction

Duplication of data in primary storage systems is quite common due to the technological trends that have been driving storage capacity consolidation. The elimination of duplicate content at both the file and block levels for improving storage space utilization is an active area of research [7, 17, 19, 22, 30, 31, 41]. Indeed, eliminating most duplicate content is inevitable in capacity-sensitive applications such as archival storage for cost-effectiveness. On the other hand, there exist systems with moderate degree of content similarity in their primary storage such as email servers, virtualized servers, and NAS devices running file and version control servers. In case of email servers, mailing lists, circulated attachments and SPAM can lead to duplication. Virtual machines may run similar software and thus create co-located duplicate content across their virtual disks. Finally, file and version control systems servers of collaborative groups often store copies of the same documents, sources and executables. In such systems, if the degree of content similarity is not overwhelming, eliminating duplicate data may not be a primary concern.

Gray and Shenoy have pointed out that given the technology trends for price-capacity and price-performance of memory/disk sizes and disk accesses respectively, disk data must "cool" at the rate of 10X per decade [11]. They suggest data replication as a means to this end. An instantiation of this suggestion is *intrinsic* replication of data created due to consolidation as seen now in many storage systems, including the ones illustrated earlier. Here, we refer to intrinsic (or application/user generated) data replication as opposed to forced (system generated) redundancy such as in a RAID-1 storage system. In such systems, capacity constraints are invariably secondary to I/O performance.

We analyzed on-disk duplication of content and I/O traces obtained from three varied production systems at FIU that included a virtualized host running two department web-servers, the department email server, and a file server for our research group. We made three observations from the analysis of these traces. First, our analysis revealed significant levels of both *disk static similarity* and *workload static similarity* within each of these systems. Disk static similarity is an indicator of the amount of duplicate content in the storage medium, while workload static similarity indicates the degree of on-disk duplicate content accessed by the I/O workload. We define these similarity measures formally in § 2. Second, we discovered a consistent and marked discrepancy between *reuse distances* [23] for sector and content in the I/O accesses on these systems indicating that content is reused more frequently than sectors. Third, there is significant overlap in content accessed over successive intervals of longer time-frames such as days or weeks.

Based on these observations, we explore the premise that intrinsic content similarity in storage systems and access to replicated content within I/O workloads can both be utilized to improve I/O performance. In doing so, we design and evaluate I/O Deduplication, a storage optimization that utilizes content similarity to either eliminate I/O operations altogether or optimize the resulting disk head movement within the storage system. I/O Deduplication comprises three key techniques: (i)

Workload type	File System size [GB]	Memory size [GB]	Reads [GB]			Writes [GB]			File System accessed
			Total	Sectors	Content	Total	Sectors	Content	
web-vm	70	2	3.40	1.27	1.09	11.46	0.86	4.85	2.8%
mail	500	16	62.00	29.24	28.82	482.10	4.18	34.02	6.27%
homes	470	8	5.79	2.40	1.99	148.86	4.33	33.68	1.44%

Table 1: **Summary statistics of one week I/O workload traces obtained from three different systems.**

content" rather than "data location" of I/O accesses in making caching decisions, (ii) *dynamic replica retrieval* that upon a cache miss for a read operation, dynamically chooses to retrieve a content replica which minimizes disk head movement, and (iii) *selective duplication* that dynamically replicates frequently accessed content in scratch space that is distributed over the entire storage medium to increase the effectiveness of dynamic replica retrieval.

We evaluated a Linux implementation of the I/O Deduplication techniques for workloads from the three systems described earlier. Performance improvements measured as the reduction in total disk busy time in the range 28-47% were observed across these workloads. We measured the influence of each technique of I/O Deduplication separately and found that each technique contributed substantially to the overall performance improvement Particularly, content-based caching increased memory caching effectiveness by at least 10% and by as much as 4X in cache hit rate for read operations. Head-position aware dynamic replica retrieval directed I/O operations to alternate locations on-the-fly and additionally reduced average I/O times by 10-20%. And finally, selective duplication created additional replicas of popular content during periods of low foreground I/O activity to further improved the effectiveness of dynamic replica retrieval, leading to a reduction in average I/O times by 23-35%. We also measured the memory and CPU overheads of I/O Deduplication and found these to be nominal.

In Section 2, we make the case for I/O deduplication. We elaborate on a specific design and implementation of its three techniques in Section 3. We perform a detailed evaluation of improvements and overhead for three different workloads in Section 4. We discuss related research in Section 5, discuss salient design and deployment alternatives in Section 6, and finally conclude with directions for future work.

2 Motivation and Rationale

In this section, we investigate the nature of content similarity and access to duplicate content using workloads from three production systems that are in active, daily use at the FIU Computer Science department. We collected I/O traces downstream of an active page cache from each system for a duration of three weeks. These systems have different I/O workloads that consist of a virtual machine running two web-servers (*web-vm* workload), an email server (*mail* workload), and a file server (*homes* workload). The *web-vm* workload is collected from a virtualized system that hosts two CS department web-servers, one hosting the department's online course management system and the other hosting the department's web-based email access portal; the local virtual disks which were traced only hosted root partitions containing the OS distribution, while the http data for these web-servers reside on a network-attached storage. The *mail* workload serves user INBOXes for the entire Computer Science department at FIU. Finally, the *homes* workload is that of a NFS server that serves the home directories of our small-sized research group; activities represent those of a typical researcher consisting of software development, testing, and experimentation, the use of graph-plotting software, and technical document preparation.

Key statistics related to these workloads are summarized in Table 1. The mail server is a heavily used system and generates a highly-intensive I/O workload in comparison to the other two. However, some uniform trends can be observed across these workloads. A fairly small percentage of the total file system data is accessed during the entire week (1.44-6.27% across the workloads), representing small working sets. Further, these are write-intensive workloads. While it is therefore important to optimize write I/O operations, we also note that most writes are committed to persistent storage in the background and do not affect user-perceived performance directly. Optimizing read operations, on the other hand, has a direct impact on user-perceived performance and system throughput because this reduces the waiting time for blocked foreground I/O operations. For read I/O's, we observe that in each workload, the unique content accessed is lesser than the unique locations that are accessed on the storage device. These observation directly motivates the three techniques of our approach as we elaborate next.

2.1 Content-based cache

The systems of interest in our work are those in which there are patterns of work shared across more than one mechanism within a single system. A *mechanism* represents any active entity, such as a single thread or process or an entire virtual machine. Such duplicated mech-

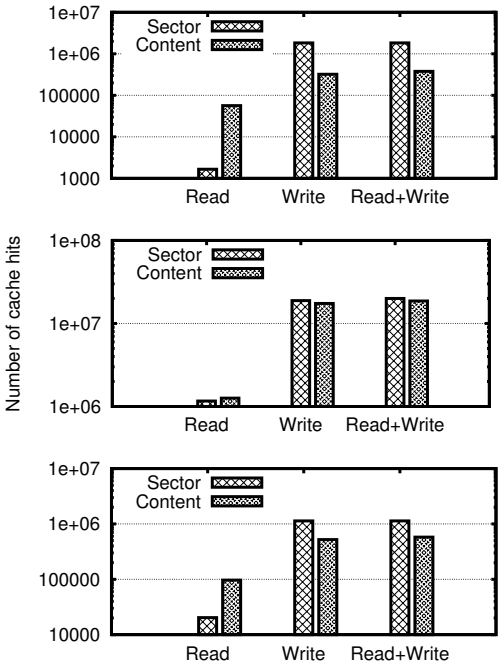


Figure 1: **Page cache hits for the web-vm (top), mail (middle), and homes (bottom) workloads.** *A single day trace was used with an infinite cache assumption.*

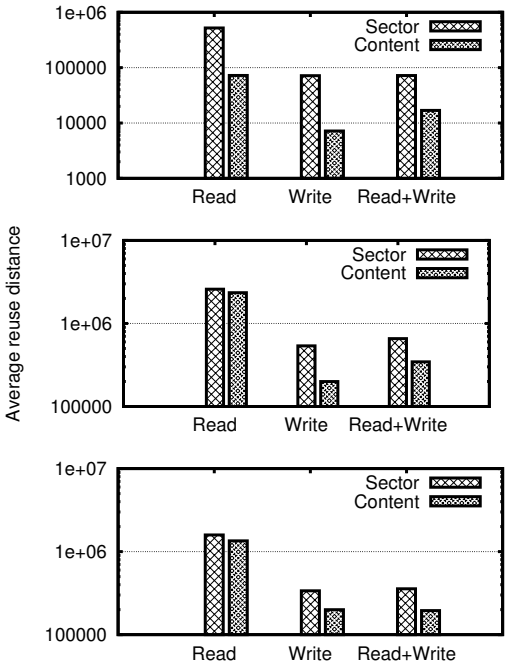


Figure 2: **Contrasting content and sector reuse distances for the web-vm (top), mail (middle), and homes (bottom) workloads.**

anisms also lead to intrinsic duplication in content accessed within the respective mechanisms' I/O operations. Duplicate content, however, may be independently managed by each mechanism and stored in distinct locations on a persistent store. In such systems, traditional storage-location (sector) addressed caching can lead to content duplication in the cache, thus reducing the effectiveness of the cache.

Figure 1 shows that cache hit ratio (for read requests) can be improved substantially by using a content-addressed cache instead of a sector-addressed one. While write I/Os leading to content hits could be eliminated for improved performance, we do not explore it in this paper. A greater number of sector hits with write I/Os are due to journaling writes by the file system, repeatedly overwriting locations within a circular journal space.

For further analysis, we define the *average sector reuse distance* for a workload as the average number of requests between successive requests to the same sector. The *average content reuse distance* is defined similarly over accesses to the same content. Figure 2 shows that the average reuse distance for content is smaller than for sector for each of the three workloads that we studied for both read and write requests. For such workloads, data addressed by content can be cache-resident for lesser time yet be more effective for servicing read requests than if the same cached data is addressed by location. Write requests on the other hand do not depend on cache hits since data is flushed to rather than requested from the storage system. These observations and those from Figure 1 motivate *content-based caching* in I/O Deduplication.

2.2 Dynamic replica retrieval

Systems with intrinsic duplication of mechanism may also operate on duplicate data stored in the persistent stores managed by each mechanism. Such intrinsic content duplication creates opportunities for optimizing I/O operations.

We define the *disk static similarity* as the average number of copies per filesystem-aligned block of content, typically of size 4KB, as a formal measure of content similarity in the storage system. The disk static similarity is calculated as $(all - zeros)/(unique - 1)$, where all is the total number of blocks, $zeroes$ are the number of zeroed blocks (never-used), and $unique$ is the number of blocks with $unique$ content (after eliminating duplicates). This static similarity measure includes blocks that are not currently in use by the file-system; we include such blocks because they were previously used and therefore may contain the same content as in-use data blocks. Table 2 summarizes static similarity values for each of the three workloads. We notice that there is substantial duplication of content on the disks used by each of these workloads. In the case of the *mail* workload, one might expect a higher level of content similarity due to

Workloads	web-vm	mail	homes
Unique pages (millions)	1.9	27	62
Total pages (millions)	5.2	73	183
Static similarity	2.67	2.64	2.94

Table 2: **Disk static similarity.** *Total pages excludes zero pages; Unique pages excludes repeated pages in addition to zero pages.*

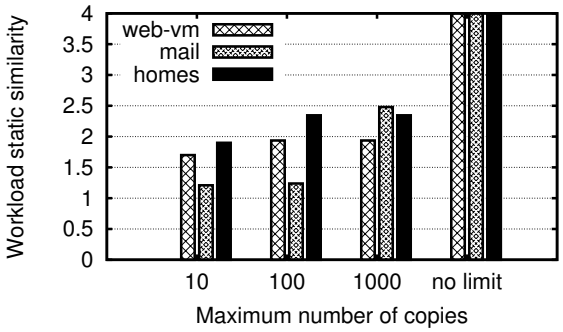


Figure 3: **Workload static similarity.** *One day traces were used. The x axis limits the static similarity consideration to blocks which have at most x copies on disk.*

mailing-list emails and circulated attachments appearing in many INBOXes. However, we point out that all emails within a user's INBOX are managed as a single large file by mail server and therefore individual emails are less likely to be aligned to the filesystem block-size, impacting the disk static similarity measure. Nevertheless, the level of content similarity in these systems is high.

While the presence of substantial duplicate content on each of these systems is promising, it is possible that duplicate content is not accessed frequently in the actual I/O workload. We measured the average number of copies in the storage system for all the blocks read within each of these workloads. We refer to this measure as the *workload static similarity*. By considering only the on-disk duplicate content pertinent to the workload we can better estimate the impact of optimizations based on content similarity. To improve the accuracy our measure, we limit the number of copies of target content. This allows us to prevent a small set of highly replicated content from inflating the workload static similarity value. As shown in Figure 3, the workload static similarity limited to content not repeated more than 1000 times is 2.5. While more than one copy of blocks read is present in the storage system on an average, we note that the disk static similarity values (in Table 2) do overestimate the performance improvement potential.

Based on these observations, we can hypothesize that for each of these workloads, accesses to data that is duplicated on the storage device can be optimally redirected

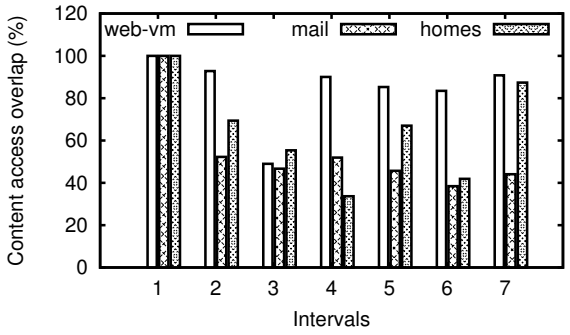


Figure 4: **Content working-sets for three week traces.** *The trace duration is divided into 7 3-day intervals and read content overlap for each interval with all content from the previous interval is presented.*

to the location that minimizes the mechanical overhead of disk I/O operations. This motivates *dynamic replica retrieval* in our approach.

2.3 Selective Duplication

A third property of workloads is repeated access to the same content. Here, we refer to accesses to specific content, which is a different measure than repeated access to the same block address. To illustrate this difference, accesses to two copies of the same executable stored within two virtual disks owned by distinct virtual machines do not lead to repeated access to the same block, but do result in repeated access to the same content.

In Figure 4, we illustrate the overlap in content being accessed across time for each of the workloads using traces over a longer, three week duration. More specifically, we divide the three week trace duration into seven, 3-day intervals and measure the overlap in content read (thus, we exclude writes) within each interval with all data accessed (both read and written) in the previous interval. The first 3-day interval uses self-similarity and therefore represents a 100% content overlap. For the remaining intervals we observe high levels of overlap in the content being read within each interval with all data accessed during the previous interval; average overlaps are 45%, 85%, and 60%, for the mail, web-vm, and homes workloads respectively.

Based on these observation, we can assume that if data accessed in the recent past were replicated in locations dispersed across the disk area, the choice in access provided by such replicas for future I/O operations can help reduce disk arm movement and improve I/O performance. Complementary findings about diurnal patterns in I/O workloads with alternating periods of low and high storage activity [8, 20] suggest that such *selective duplication*, if performed opportunistically during night-time, may result in negligible impact to foreground I/O activity.

3 System Design

I/O Deduplication systematically explores the use of content similarity within storage systems to reduce the mechanical delays incurred in I/O operations and/or to eliminate I/O operations altogether. In this section, we start with an overview of the system architecture and then present the various design choices and rationale behind constructing each of the three mechanisms that constitute I/O Deduplication.

3.1 Architectural Overview

An optimization based on content similarity can be built at various layers of the storage stack, with varying degrees of access and control over storage devices and the I/O workload. Prior research has argued for building storage optimizations in the block layer of the storage stack [12]. We choose the block layer for several reasons. First, the block interface is a generic abstraction that is available in a variety of environments including operating system block device implementations, software RAID drivers, hardware RAID controllers, SAN (e.g., iSCSI) storage devices, and the increasingly popular storage virtualization solutions (e.g., IBM SVC [16], EMC Invista [9], NetApp V-Series [28]). Consequently, optimizations based on the block abstraction can potentially be ported and deployed across these varied platforms. In the rest of the paper, we develop an operating system block device oriented design and implementation of I/O Deduplication. Second, the simple semantics of block layer interface allows easy I/O interception, manipulation, and redirection. Third, by operating at the block layer, the optimization becomes independent of the file system implementation, and can support multiple instances and types of file systems. Fourth, this layer enables simplified control over system devices at the block device abstraction, allowing an elegantly simple implementation of selective duplication that we describe later. Finally, additional I/Os generated by I/O Deduplication can leverage I/O scheduling services, thereby automatically addressing the complexities of block request merging and reordering.

Figure 5 presents the architecture of I/O Deduplication for a block device in relation to the storage stack within an operating system. We augment the storage stack's block layer with additional functionality, which we term *I/O Deduplication layer*, to implement the three major mechanisms: the content-based cache, the dynamic replica retriever, and the selective duplicator. The *content-based cache* is the first mechanism encountered by the I/O workload which filters the I/O stream based on hits in a content-addressed cache. The *dynamic replica retriever* subsequently optionally redirects the unfiltered read I/O requests to alternate locations on the disk to avail the best access latencies to requests. The *selective duplicator* is composed of a *kernel* sub-component that tracks content accesses to create a candidate list of content for replication, and a *user-space* process that runs during periods of low disk activity and populates replica content in scratch space distributed across the entire disk. Thus, while the kernel components run continuously, the user-space component runs sporadically. Separating out the actual replication process into a user-level thread allows greater user/administrator control over the timing and resource consumption of the replication process, an I/O resource-intensive operation. Next, we elaborate on the design of each of the three mechanisms within I/O Deduplication.

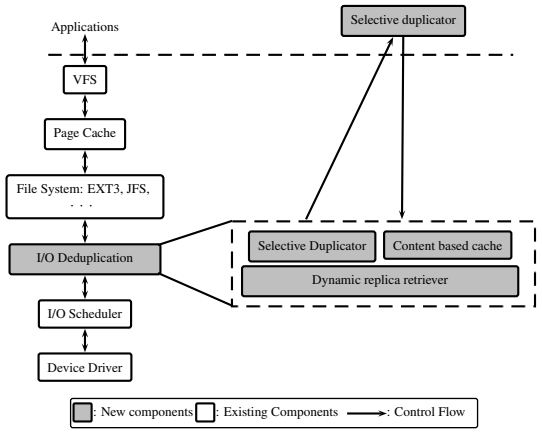


Figure 5: **I/O Deduplication System Architecture.**

3.2 Content based caching

Building a content based cache at the block layer creates an additional buffer cache separate from the virtual file system (VFS) cache. Requests to the VFS cache are sector-based while those to the I/O Deduplication cache are both sector- and content-based. The I/O Deduplication layer only sees the read requests for sector misses in the VFS cache. We discuss exclusivity across these caches shortly. In the I/O Deduplication layer, read requests identified by sector locations are queried against a dual sector- and content-addressed cache for hits before entering the I/O scheduler queue or being merged with an existing request by the I/O scheduler. Population of the content-based cache occurs along both the read and write paths. In case of a cache miss during a read operation, the I/O completion handler for the read request is intercepted and modified to additionally insert the data read into the content-addressed cache after I/O completion only if it is not already present in the cache and is important enough in the LRU list to be cached. A write request to a sector which had contained duplicate data is simply removed from the corresponding duplicate sector list to ensure data consistency for future accesses. The new data contained within write requests is optionally

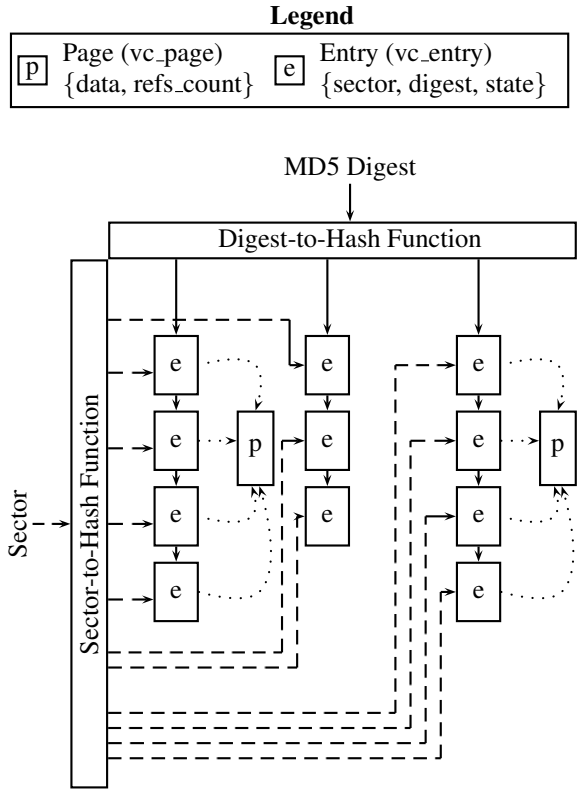


Figure 6: **Data structure for the content-based cache.** *The cache is addressable by both sector and content-hash. vc_entrys are unique per sector. Solid lines between vc_entrys indicates that they may have the same content (they may not in case of hash function collisions.) Dotted lines form a link between a sector (vc_entry) and a given page (vc_page.) Note that some vc_entrys do not point to any page – there is no cached content cached for these. However, this indicates that the linked vc_entrys have the same data on disk. This happens when some of the pages are evicted from the cache. Additionally, pages form an LRU list.*

inserted into the content-addressed cache (if it is sufficiently important) in the onward path before entering the request into the I/O scheduler queue to keep the content cache up-to-date with important data.

The in-memory data structure implementing the content-based cache supports look-up based on both sector and content-hash to address read and write requests respectively. Entries indexed by content-hash values contain a sector-list (list of sectors in which the content is replicated) and the corresponding data if it was entered into the cache and not replaced. Cache replacement only replaces the content field and retains the sector-list in the in-memory content-cache data structure. For read requests, a sector-based lookup is first performed to determine if there is a cache hit. For write requests, a content-hash based look-up is performed to determine a hit and the sector information from the write request is added to the sector-list. Figure 6 describes the data structure used to manage the content-based cache. A write to a sector that is present in a sector-list indexed by content-hash is simply removed from the sector list and inserted into a new list based on the sector's new content hash. It is important to also point out that our design uses a write-through cache to preserve the semantics of the block layer. Next, we discuss some practical considerations for our design.

Since the content cache is a second-level cache placed below the file system page cache or, in case of a virtualized environment, within the virtualization mechanism, typically observed recency patterns in first level caches are lost at this caching layer. An appropriate replacement algorithm for this cache level is therefore one that captures frequency as well. We propose using Adaptive Replacement Cache (ARC) [24] or CLOCK-Pro [18] as good candidates for a second-level content-based cache and evaluate our system with ARC and LRU for contrast.

Another concern is that there can be a substantial amount of duplicated content across the cache levels. There are two ways to address this. Ideally, the content-based cache should be integrated into a higher level cache (e.g., VFS page cache) implementations if possible. However, this might not be feasible in virtualized environments where page caches are managed independently within individual virtual machines. In such cases, techniques that help make in-memory cache content across cache levels exclusive such as cache hints [21], demotions [38], and promotions [10] may be used. An alternate approach is to employ memory deduplication techniques such as those proposed in the VMware ESX server [36], Difference Engine [13], and Satori [25]. In these solutions, duplicate pages within and across virtual machines are made to point to the same machine frame with use of an extra level of indirection such as the shadow page tables. In memory duplicate content across multiple levels of caches is indeed an orthogonal problem and any of the referenced techniques could be used as a solution directly within I/O Deduplication.

3.3 Dynamic replica retrieval

The design of dynamic replica retrieval is based on the rationale that better I/O schedules can be constructed with more options for servicing I/O requests. A storage system with high disk static similarity (i.e., duplicated content) creates such options naturally. With dynamic replica retrieval in such a system, read I/O requests are optionally indirected to alternate locations before entering the I/O scheduler queue. Choosing alternate locations for write requests is complicated due to the need for ensuring up-to-date block content; while we do not con-

sider this possibility further in our work, investigating alternate mechanisms for optimizing write operations to utilize content similarity is certainly a promising area of future work. The content-addressed cache data structure that we explored earlier supports look-up based on sector (contained within a read request) and returns a sector-list that contain replicas of the requested content, thus providing alternate locations to retrieve the data from.

To help decide if and to where a read I/O request should be redirected, the dynamic replica retriever continuously maintains an estimate of the disk head position by monitoring I/O completion events. For estimating head position, we use read I/O completion events only and ignore I/O completion events for write requests since writes may be reported as complete as soon as they are written to the disk cache. Consequently, the head position as computed by the dynamic replica retriever is an approximation, since background write flushes inside the disk are not accounted for. To implement the head-position estimator, the last head position is updated during the execution of the I/O completion handler of each read request. Additionally, the direction of the disk arm managed by the scheduler is also maintained for elevator-based I/O schedulers.

One complication with redirection of an I/O request before a possible merge operation (done by the I/O scheduler later) is that this optimization can reduce the chances for merging the request with another request already awaiting service in the I/O scheduler queue. For each of the workloads we experimented with, we did indeed observe reduction in merging negatively affecting performance when using redirection purely based on current head-position estimates. Request merging should gain priority over any other operation since it eliminates mechanical overhead altogether. One means to prioritize request merging is performing the indirection of requests below the I/O scheduler which performs merging within its mechanisms. Although this is an acceptable and correct solution, it is substantially more complex compared to implementation at the block layer above the I/O scheduler because there are typically multiple dispatch points for I/O scheduler implementations inside the operating system. The second option, and the one used in our system, is to evaluate whether or not to redirect the I/O request to a more opportune location, based on the an actively maintained digest of outstanding requests at the I/O scheduler – these are requests that have been dispatched to the I/O scheduler but not yet reported as completed by the device. If an outstanding request to a location adjacent to the current request exists in the digest, redirection is avoided to allow for merging.

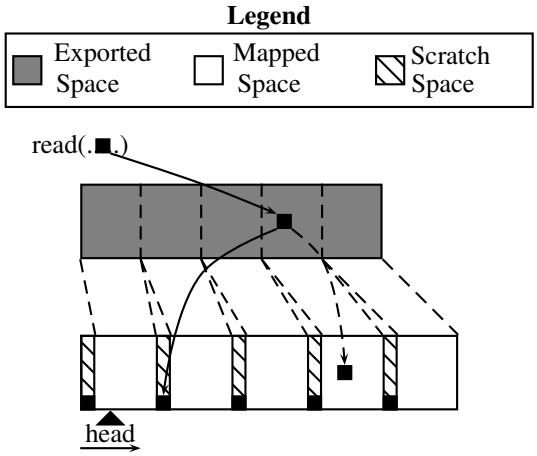


Figure 7: **Transparent replica management for selective duplication.** *The read request to the solid block in the exported space can either be retrieved from its original location in the mapped space or from any of the replicas in the scratch space that reduce head movement.*

3.4 Selective duplication

Figure 4 revealed that the overlap in longer-time frame working sets can be substantial in workloads, more than 80% in some cases. While such overlapping content are the perfect choice for content to be cached, such content was found to be too big to fit in memory.

A complementary optimization to dynamic replica retrieval based on this observation is that an increase in the number of duplicates for popular content on the disk can create even greater opportunities for optimizing the I/O schedule. A basic question then is *what* to duplicate and *when*. We implemented *selective duplication* to run every day during periods of low disk activity based on the observed diurnal patterns in the I/O workloads that we experimented with. The question of what to duplicate can be rephrased as what is the content accessed in the previous days that is likely to be accessed in the future? Our analysis of the workloads revealed that the content overlap between the most frequently used content of the previous days was found to be a good predictor of future accesses to content. The selective duplicator kernel component calculates the list of frequently used content across multiple days by extending the ARC replacement algorithm used for the content-addressed cache.

A list of sectors to duplicate is then forwarded to the user-space replicator process which creates the actual replicas during periods of low activity. The periodic nature of this process ensures that the most relevant content is replicated in the scratch space while older replicas of content that have either been overwritten or are no longer important are discarded. To make the replication process seamless to file system, we implemented *trans-*

parent replica management that implements the scratch space used to store replicas transparently. The scratch space is provisioned by creating additional physical storage volumes/partitions interspersed within the file system data. Figure 7 depicts the transparent replica management wherein the storage is interspersed with five scratch space volumes interspersed between file system mapped space. For file system transparency, a single logically contiguous volume is presented to the file system by the I/O Deduplication extension. The scratch space is used to create one or more replicas of data in the exported space. Since the I/O operations issued during the selective duplication process are themselves routed via the in-kernel I/O Deduplication components, the additional content similarity information due to replication is automatically recorded into the content cache.

3.5 Persistence of metadata

A final issue is the persistence of the in-memory data structure so that the system can retain intelligence about content similarity across system restart operations. Persistence is important for retaining the locations of on-disk intrinsic and artificially created duplicate content so that this information can be restored and used immediately upon a system restart event. We note that while persistence is useful to retain intelligence that is acquired over a period of time, "continuous persistence" of metadata in I/O Deduplication is not necessary to guarantee the reliability of the system, unlike other systems such as the eager writing disk array [40] or doubly distorted mirroring [29]. In this sense, *selective duplication* is similar to the opportunistic replication as performed by FS2 [15] because it tracks updates to replicated data in memory and only guarantees that the primary copy of data blocks are up-to-date at any time. While persistence of the in-memory data is not implemented in our prototype yet, guaranteeing such persistence is relatively straightforward. Before the I/O Deduplication kernel module is unloaded (occuring at the same time the managed file system is unmounted), all in-memory data structure entries can be written to a reserved location of the managed scratch-space. These can then be read back to populate the in-memory metadata upon a system restart operation when the kernel module is loaded into the operating system.

4 Experimental Evaluation

In this section, we evaluate each mechanism in I/O Deduplication separately first and then evaluate their cumulative performance impact. We also evaluate the CPU and memory overhead incurred by an I/O Deduplication system. We used the block level traces for the three systems that were described in detail in § 2 for our evaluation. The traces were replayed as block traces in a similar way as done by blktrace [2]. Blktrace could not be used as-is since it does not record content information; we used a custom Linux kernel module to record content-hashes for each block read/written in addition to other attributes of each I/O request. Additionally, the blktrace tool btreplay was modified to include traces in our format and replay them using provided content. Replay was performed at a maximum acceleration of 100x with care being taken in each case to ensure that block access patterns were not modified as a result of the speedup. Measurements for actual disk I/O times were obtained with per-request block-level I/O tracing using blktrace and the results reported by it. Finally, all trace playback experiments were performed on a single Intel(R) Pentium(R) 4 CPU 2.00GHz machine with 1 GB of memory and a Western Digital disk WD5000AAKB-00YSA0 running Ubuntu Linux 8.04 with kernel 2.6.20.

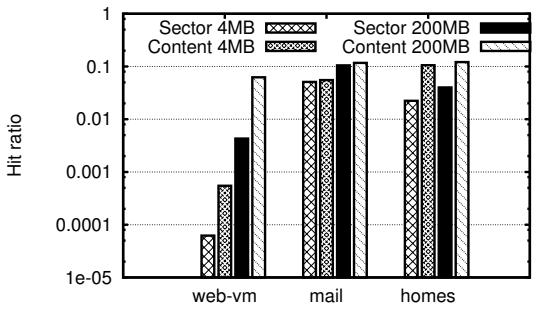


Figure 8: **Per-day page cache hit ratio for content- and sector- addressed caches for read operations.** *The total number of pages read are 0.18, 2.3, and 0.23 million respectively for the web-vm, mail and homes workloads. The numbers in the legend next to each type of addressing represent the cache size.*

4.1 Content based cache

In our first experiment, we evaluated the effectiveness of a content-addressed cache against a sector-addressed one. The primary difference in implementation between the two is that for the sector-addressed cache, the same content for two distinct sectors will be stored twice. We fixed the cache size in both variants to one of two different sizes, 1000 pages (4MB) and 50000 pages (200MB). We replayed two weeks of the traces for each of the three workloads; the first week warmed up the cache and measurements were taken during the second week. Figure 8 shows the average per-day cache hit counts for read I/O operations during the second week when using an *adaptive replacement cache* (ARC) in two modes, content and sector addressed.

This experiment shows that there is a large increase in per-day cache hit counts for the web and the home work-

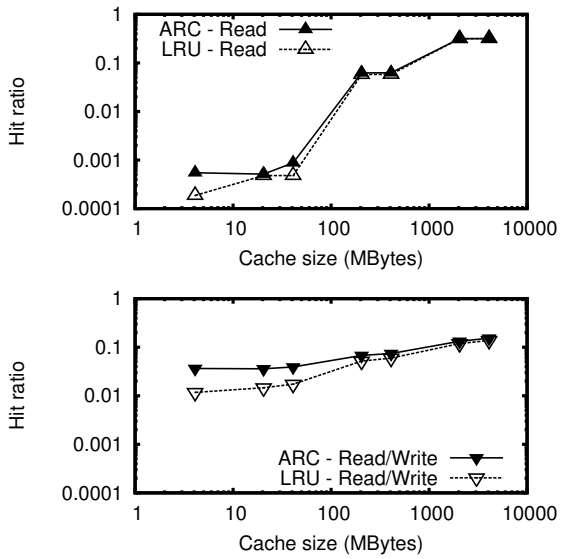


Figure 9: **Comparison of ARC and LRU content based caches for pages read only (top) and pages read/write operations (bottom).** *A single day trace (0.18 million page reads and 2.09 million page read/writes) of the web workload was used as the workload.*

loads when a content-addressed cache is used (relative to a sector-addressed cache). The first observation is that improvement trends are consistent across the two cache sizes. Both caches implementations benefit substantially from a larger cache size except for the *mail* workload, indicating that *mail* is not a cache-friendly workload validated by its substantially larger working set and workload I/O intensity (as observed in Section 2). The *web-vm* workload shows the biggest increase with an almost 10X increase in cache hits with a cache of 200MB compared to the home workload which has an increase of 4X. The *mail* workload has the least improvement of approximately 10%.

We performed additional experiments to compare an LRU implementation with the ARC cache implementation (used in the previous experiments) using a single day trace of the *web-vm* workload. Figure 9 provides a performance comparison of both replacement algorithms when used for a content-addressed cache. For small and large cache sizes, we observe that ARC is either as good or more effective than LRU with ARC's improvement over LRU increasing substantially for write operations at small to moderate cache sizes. More generally, this experiment suggests that the performance improvements for a content-addressed cache are sensitive to the cache replacement mechanism which should be chosen with care.

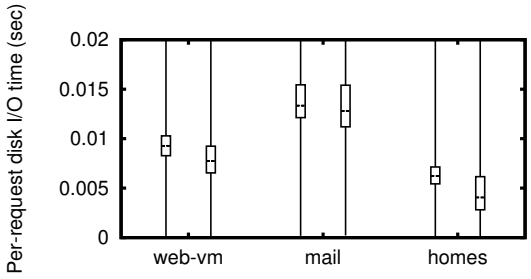


Figure 10: **Improvement in disk read I/O times with dynamic replica retrieval.** *Box and whisker plots depicting median and quartile values of the per-request disk I/O times are shown. For each workload, the values to the left represent the vanilla system and that on the right is with dynamic replica retrieval.*

4.2 Dynamic replica retrieval

To evaluate the effectiveness of dynamic replica retrieval, we replayed a one week trace for each workload with and without using I/O Deduplication. When using I/O Deduplication, prior to replaying the trace workload, information about duplicates was loaded into the kernel module's data structures, as would have been accumulated by I/O Deduplication over the lifetime of all data on the disk. Content-based caching and selective duplication were turned-off. In each case, we measured the per-request disk I/O time per request. A lower per-request disk I/O time informs us of a more efficient storage system.

Figure 10 shows the results of this experiment. For all the workloads there is a decrease in median per-request disk I/O time of at least 10% and up to 20% for the homes workload. These findings indicate that there is room for optimizing I/O operations simply by using pre-existing duplicate content on the storage system.

4.3 Selective duplication

Given the improvements offered by dynamic replica retrieval, we now evaluate the impact of selective duplication, a mechanism whose goal is to further increase the opportunities for dynamic replica retrieval. The workloads and metric used for this experiment were the same as the ones in the previous experiment.

To perform selective duplication, for each workload, ten copies of the predicted popular content were created on scratch space distributed across the entire disk drive. The set of popular data blocks to replicate is determined by the kernel module during the day and exported to user space after a time threshold is reached. A user space program logs the information about the popular content that are candidates for selective duplication and creates the copies on disk based on the information gathered during periods of little or no disk activity. As in the previous

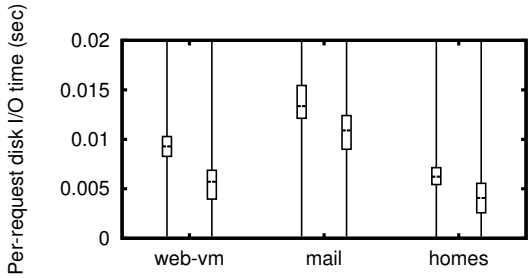


Figure 11: **Improvement in disk read I/O times with selective duplication and dynamic replica retrieval optimizations.** *Other details are the same as Figure 10.*

experiment, prior to replaying the trace workload, all the information about duplicates on disk was loaded into the kernel module's data structures.

Figure 11 (when compared with the numbers in Figure 10) shows how selective duplication improves upon the previous results using pure dynamic replica retrieval. Figure 4 showed that the web workload had more than 80% in content reuse overlap and the effect of duplicating this information can be observed immediately. Overall, the reduction in per-request disk I/O time was improved substantially for the *web-vm* and *homes* workloads, and to a lesser extent for the *homes* workload using this additional technique when compared to using dynamic replica retrieval alone. Overall reductions in median disk I/O times when compared to the vanilla system were 33% for the web workload, 35% for the homes workload, and 23% for mail.

4.4 Putting it all together

We now examine the impact of using all the three mechanisms of I/O Deduplication at once for each workload. We use a sector-addressed cache for the baseline *vanilla* system and a content-addressed one for *I/O Deduplication*. We set the cache size to 200 MB in both cases. Since sector- or content-based caching is the first mechanism encountered by the I/O request stream, the results of the caching mechanism remain unaffected because of the other two, and the cache hit counts remain as with the independent measurements reported in Section 4.1. However, cache hits do modify the request stream presented to the remaining two optimizations. While there is a reduction in the improvements to per-request disk read I/O times with all three mechanisms (not shown) when compared to using the combination of dynamic replica retrieval and selective duplication alone, the total number of I/O requests is different in each case. Thus the average disk I/O time is not a robust metric to measure relative performance improvement. The total disk read I/O time for a given I/O workload, on the other hand, provides an accurate comparative evaluation by taking into account both the reduced number of I/O read operations

Workload	Vanilla (sec)	I/O dedup (sec)	Improvement
web-vm	3098.61	1641.90	47%
mail	4877.49	3467.30	28%
home	1904.63	1160.40	39%

Table 3: **Reduction in total disk read I/O times.**

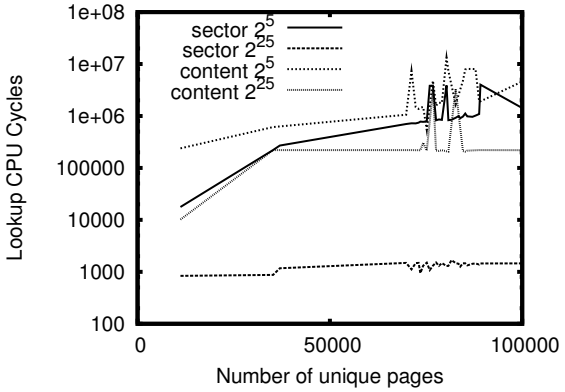


Figure 12: **Overhead of content and sector lookup operations with increasing size of the content-based cache.**

due to content-based caching and the improvements in disk latencies of the latter two optimizations, and effectively measures the true increase in disk I/O efficiency.

When comparing total disk read I/O time for these three workloads, substantial reductions were observed when compared to a vanilla system as shown on Table 3. These uniformly large improvements (28-47% across the three workloads) are a clear indication of the effectiveness of I/O Deduplication in improving I/O performance for a range of different storage workloads.

4.5 Evaluating Overhead

While the gains due to I/O Deduplication are promising, it incurs resource overhead. Specifically, the implementation uses content- and sector- addressed hash-tables to simplify lookup and insert operations into the content based cache. We evaluate the CPU overhead for insert/lookup operations and memory overhead required for managing hash-table metadata in I/O Deduplication.

4.5.1 CPU Overhead

To evaluate the overhead of I/O Deduplication, we measured the average number of CPU cycles required for lookup/insert operations as we vary the number of unique pages (i.e., size) in the content-based cache (i.e., cache size) for a day of the web workload. Figure 13 depicts these overheads for two cache configurations, one configured with 2^{25} buckets in the hash tables and the other with 2^5 buckets. Read operations perform a sector lookup and additionally content lookup in case of a miss

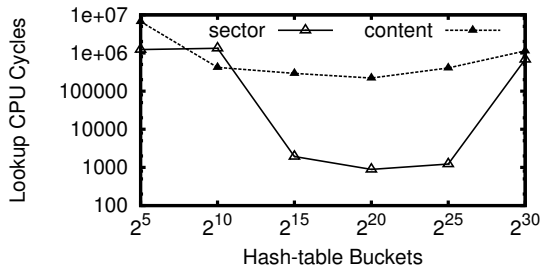


Figure 13: **Overhead of sector and content lookup operations with increasing hash-table bucket entries.**

for insertion. Write operations always perform a sector and content lookup due to our write-through cache design. Content lookups need to first compute the hash for the page contents which takes around 100000 CPU cycles for MD5. With few buckets (2^5) lookup times approach O(N) where N is the size of the hash-table. However, given enough hash-table buckets (2^{25}), lookup times are O(1).

Next, we examined the sensitivity to the hash-table bucket entries. As the number of buckets are increased, the lookup times decrease as expected due to reduction in collisions, but beyond 2^{20} buckets, there is an increase. We attribute this to L2 cache and TLB misses due to memory fragmentation, under-scoring that hash-table bucket sizes should be configured with care. In the sweet spot of bucket entries, the lookup overhead for both sector and content reduces to 1K CPU cycles or less than 1μs for our 2GHz machine. Note that the content lookup operation includes a hash computation which inflates its cycles requirement by at least 100000.

4.6 Memory Overhead

The management of I/O Deduplication's content-based cache introduces memory overhead for managing metadata for the content-based cache. Specifically, the memory overhead is dictated by the size of the cache measured in pages (P), the degree of *Workload static similarity* (WSS), and the configured number of buckets in the hash tables (HTB) which also determine the lookup time as we saw earlier. In our current unoptimized implementation, the memory overhead in bytes (assuming 4 bytes pointers and 4096 bytes pages) :

$$mem(P, WSS, HTB) = 13 * P + 36 * P * WSS + 8 * HTB \quad (1)$$

These overheads include 13 bytes per-page to store the metadata for a a specific page content (*vc_page*), 36 bytes per page per duplicated entry (*vc_entry*), and 8 bytes per hash-table entry for the corresponding linked list. For a 1GB content cache (256K pages), a static similarity of 4, and a hash-table of size 1 million entries, the metadata overhead is ∼48MB or approximately 4.6%.

5 Related Work

In this section, we examine research literature related to workload-based I/O performance optimization and research related to the use of content similarity in memory and storage systems. While there is substantial work done along both these directions, they are for the most part explored as orthogonal techniques in the literature, with the latter primarily being used for optimizing storage capacity utilization using data deduplication.

5.1 I/O performance optimization

Workload-based I/O performance optimization has a long history. The first class of optimizations is based on creating optimized layouts for storage system data. The early works of Wong [37], Vongsathorn *et al.* [35], and Ruemmler and Wilkes [32], which argued for shuffling on-disk data based on data access frequency. Later, Akyurek and Salem [1] argued for copying over shuffling of data with the observation that original layouts are often useful and data popularity and access patterns can be temporary. More recently, ALIS [14] and BORG [3] have employed a dedicated, reorganized area on the disk to improve both locality and sequentiality of I/O access.

The second class of work is based on replicating data and creating opportunities for reducing disk head movement by increasing the number of choices for retrieving data. These include the large body of work on mirroring systems [4]. The work on doubly distorted mirrors [33] creates multiple replicas on master and slave disks to increase both write performance (using initial write-anywhere and background updating of original locations) and read performance by dispatching read requests to the nearest free arm. Zhang *et al.*'s work on eager writing [40] extended this approach to mirrored/striped RAID configurations primarily for database OLTP workload (which are characterized by little locality or sequentiality). Yu *et al.* [39] propose an alternate approach for trading disk capacity for performance in a RAID system, by storing several *rotational replicas* of each block and using a rotational latency sensitive disk scheduler. FS2 [15] proposed replication in file system free-space based on block-access frequency and the use of such selective duplication of content to optimize head movement during subsequent retrieval of replicated data. Quite obviously, selective duplication is motivated by the above works, but is different in two respects: *(i)* it targets identifying replication candidates based on content popularity, rather than block address popularity, and *(ii)* duplication is performed in pre-configured dedicated space transparently to the file system and/or other managers of the storage system. To the best of our knowledge the only work to use content-based optimization of I/O is the work of Tolia *et al.* [34], where the authors use content hashes to perform dynamic replica retrieval choosing be-

tween multiple hosts in an extrinsically-duplicated distributed storage system. Our work, on the other hand, uses intrinsic duplication within a single storage system.

5.2 Data deduplication

Content similarity in both memory and archival storage have been investigated in the literature. Memory deduplication has been explored before in the VMware ESX server [36], Difference Engine [13], and Satori [25], each aiming to eliminate duplicate in-memory content both within and across virtual machines sharing a physical host. Of these, Satori has apparent similarities to our work because it identifies candidates for in-memory deduplication as data is read from storage. Satori runs in two modes: content-based sharing and copy-on-write disk sharing. For content-based sharing, Satori uses content-hashes to track page contents in memory read from disk. Since its goal is not I/O performance optimization, it does not track duplicate sectors on disk and therefore does not eliminate duplicated I/Os that would read the same content from multiple locations. In copy-on-write disk sharing, the disk is already configured to be copy-on-write enabling the sharing of multiple VM disk images on storage. In this mode, duplicated I/Os due to multiple VMs retrieving the same sectors on the shared physical disk would be eliminated in the same way as a regular sector-addressed cache would do. In contrast, our work targets I/O performance optimization by either eliminating I/Os if it were to retrieve duplicate content irrespective of where it may reside on storage or reducing head movement otherwise Thus, the contributions of Satori are complementary to our work and can be used simultaneously.

Data deduplication in archival storage has also gained importance in both the research and industry communities. Current research on data deduplication uses several techniques to optimize the I/O overheads incurred due to data duplication. Venti [30] proposed by Quinlan and Dorward was the first to propose the use of a content-addressed storage for performing data deduplication in an archival system. The authors suggested the use of an in-memory content-addressed index of data to speed up lookups for duplicate content. Similar content-addressed caches were used in data backup solutions such as Peabody [26] and Foundation [31]. Content-based caching in I/O Deduplication is inspired by these works. Recent work by Zhu and his colleagues [41] suggests new approaches to alleviate the disk bottleneck via the use of Bloom filters [5] and by further accounting for locality in the content stream. The Foundation work suggests additional optimizations using batched retrieval and flushing of index entries and a log-based approach to writing data and index entries to utilize temporal locality [31]. The work on sparse indexing [22] suggests improvements to Zhu *et al.*'s general approach by exploiting locality in the chunk index lookup operations to further mitigate the disk I/O bottleneck. I/O Deduplication addresses a orthogonal problem, that of improving I/O performance for foreground I/O workload based on the use of duplicates, rather than their elimination. Nevertheless, the above approaches do suggest interesting techniques to optimize the management of a content-addressed index and cache in main-memory that is complementary to and can be used directly within I/O Deduplication.

6 Discussion

Several aspects of I/O Deduplication from design, implementation, and deployment standpoints warrant further discussion. Some of these also suggest avenues for future work.

Multi-disk deployment. In previous sections, we designed and evaluated a single disk implementation of I/O Deduplication. Multi-disk storage deployments in the form of RAID or more complex NAS appliances are common in enterprise data centers. One might question both the utility and effectiveness of the single disk head movement optimizations central to I/O Deduplication in such systems. We believe that head movement optimizations based on content similarity is viable and can enable complementary optimizations by minimizing the unavoidable mechanical delays in any disk-based storage system. The dynamic replica retrieval and selective duplication sub-techniques require further consideration for multi-disk systems. First, these optimizations must be implemented where information about individual disk head positions is available. Such information is available inside the driver for software RAID, in the RAID controller for hardware RAID, and inside the firmware/OS or internal hardware controllers for NAS appliances. Digest information about the outstanding requests and I/O completion events at each disk can then be utilized as in the single disk design. While the optimal location within each disk for each I/O request can be thus compiled, the complementary issue of load balancing across multiple disks must also be addressed. Apart from the well-known queue depth based techniques for load-balancing, alternate solutions such as simultaneous dispatching to multiple disks combined with just-in-time I/O cancellation can also be envisioned where applicable.

Hash collisions. Our design and implementation of I/O Deduplication makes the assumption that MD5 (128 bits) is collision free. Specifically, this assumption is made when the content-hash entry for a new page being written is registered. A similar assumption, for SHA-1 is made for deduplication in archival storage [30] and low-bandwidth network file transfers [27]. While this as-

sumption may be reasonable in several settings, delivering absolute correctness guarantees requires that this assumption be removed. Systems like Foundation [31] additionally include the provision to perform a byte-wise comparison following a hit in the content cache by reading the target location which potentially contains the duplicate data. This, of course, requires an additional I/O operation. The use of a specific hash function or the method of determining duplicate content is not decisive in our design, and these alternatives can be employed if found necessary within the target deployment scenario.

Variable-sized chunks. Our implementation of I/O Deduplication uses fixed size blocks as the basic data unit for determining content similarity. This choice was motivated by our goal of simplified deployment on a variety of block storage systems. Using variable size chunks as units has been demonstrated to be more effective for similarity detection for mostly similar content and similar content at different offsets within a file [6, 27]. This capability is especially important for archival storage where a single backup file is composed of multiple data files stored at different offsets and possibly with partial modifications. We believe that for online storage systems, this may be of lesser concern, except for very specific applications (e.g., a mail server where entire user INBOXes or folders are managed as a single file). Nevertheless, the use of variable sized chunks for I/O deduplication provides an interesting avenue of future work.

7 Conclusions and Future work

System and storage consolidation trends are driving increased duplication of data within storage systems. Past efforts have been primarily directed towards the elimination of such duplication for improving storage capacity utilization. With I/O Deduplication, we take a contrary view that intrinsic duplication in a class of systems which are not capacity-bound can be effectively utilized to improve I/O performance – the traditional Achilles' heel for storage systems. Three techniques contained within I/O Deduplication work together to either optimize I/O operations or eliminate them altogether. An in-depth evaluation of these mechanisms revealed that together they reduced average disk I/O times by 28-47%, a large improvement all of which can directly impact the overall application-level performance of disk I/O bound systems. The content-based caching mechanism increased memory caching effectiveness by increasing cache hit rates by 10% to 4x for read operations when compared to traditional sector-based caching. Head-position aware dynamic replica retrieval directed I/O operations to alternate locations on-the-fly and additionally reduced I/O times by 10-20%. And, selective duplication created additional replicas of popular content during periods of low foreground I/O activity and further improved the effectiveness of dynamic replica retrieval by 23-35%.

I/O Deduplication opens up several directions for future work. One avenue for future work is to explore content-based optimizations for write I/O operations. A possible future direction is to optionally coalesce or even eliminate altogether write I/O operations for content that are already duplicated elsewhere on the disk, or alternatively direct such writes to alternate locations in the scratch space. While the first option might seem similar to data deduplication at a high-level, we suggest a primary focus on the performance implications of such optimizations rather than capacity improvements. Any optimization for writes affects the read-side optimizations of I/O Deduplication and a careful analysis and evaluation of the trade-off points in this design space is important.

Acknowledgments

We thank the anonymous reviewers and our shepherd Ajay Gulati for excellent feedback which improved this paper substantially. We thank Eric Johnson for his help with production server traces at FIU. This work was supported by the NSF grants CNS-0747038 and IIS-0534530 and by DoE grant DE-FG02-06ER25739.

References

[1] Sedat Akyurek and Kenneth Salem. Adaptive Block Rearrangement. *Computer Systems*, 13(2):89–121, 1995.

[2] Jens Axboe. blktrace user guide, February 2007.

[3] Medha Bhadkamkar, Jorge Guerra, Luis Useche, Sam Burnett, Jason Liptak, Raju Rangaswami, and Vagelis Hristidis. BORG: Block-reORGanization for Self-optimizing Storage Systems. In *Proc. of the USENIX File and Storage Technologies*, February 2009.

[4] Dina Bitton and Jim Gray. Disk Shadowing. In *Proc. of the International Conference on Very Large Data Bases*, 1988.

[5] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[6] Sergey Brin, James Davis, and Hector Garcia-Molina. Copy Detection Mechanisms for Digital Documents. In *Proc. of ACM SIGMOD*, May 1995.

[7] Austin Clements, Irfan Ahmad, Murali Vilayannur, and Jinyuan Li. Decentralized deduplication in san cluster file systems. In *Proc. of the USENIX Annual Technical Conference*, June 2009.

[8] Daniel Ellard, Jonathan Ledlie, Pia Malkani, and Margo Seltzer. Passive NFS Tracing of Email and Research Workloads. In *Proc. of the USENIX Conference on File and Storage Technologies*, March 2003.

[9] EMC Corporation. EMC Invista. http://www.emc.com/products/software/invista/invista.jsp.

[10] Binny S. Gill. On multi-level exclusive caching: offline optimality and why promotions are better than demotions.

In *Proc. of the USENIX File and Storage Technologies*, Feburary 2008.

[11] Jim Gray and Prashant Shenoy. Rules of Thumb in Data Engineering. *Proc. of the IEEE International Conference on Data Engineering*, February 2000.

[12] Jorge Guerra, Luis Useche, Medha Bhadkamkar, Ricardo Koller, and Raju Rangaswami. The Case for Active Block Layer Extensions. *ACM Operating Systems Review*, 42(6), October 2008.

[13] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey Voelker, and Amin Vahdat. Difference Engine: Harnessing Memory Redundancy in Virtual Machines. *Proc. of the USENIX OSDI*, December 2008.

[14] Windsor W. Hsu, Alan Jay Smith, and Honesty C. Young. The Automatic Improvement of Locality in Storage Systems. *ACM Transactions on Computer Systems*, 23(4):424–473, Nov 2005.

[15] Hai Huang, Wanda Hung, and Kang G. Shin. FS2: Dynamic Data Replication In Free Disk Space For Improving Disk Performance And Energy Consumption. In *Proc. of the ACM SOSP*, October 2005.

[16] IBM Corporation. IBM System Storage SAN Volume Controller. http://www-03.ibm.com/systems/storage/software/virtualization/svc/.

[17] N. Jain, M. Dahlin, and R. Tewari. TAPER: Tiered Approach for Eliminating Redundancy in Replica Synchronization. In *Proc. of the USENIX Conference on File And Storage Systems*, 2005.

[18] Song Jiang, Feng Chen, and Xiaodong Zhang. Clock-pro: An effective improvement of the clock replacement. In *Proc. of the USENIX Annual Technical Conference*, April 2005.

[19] P. Kulkarni, F. Douglis, J. D. LaVoie, and J. M. Tracey. Redundancy Elimination Within Large Collections of Files. *Proc. of the USENIX Annual Technical Conference*, 2004.

[20] Andrew Leung, Shankar Pasupathy, Garth Goodson, and Ethan Miller. Measurement and Analysis of Large-Scale Network File System Workloads. *Proc. of the USENIX Annual Technical Conference*, June 2008.

[21] Xuhui Li, Ashraf Aboulnaga, Kenneth Salem, Aamer Sachedina, and Shaobo Gao. Second-tier cache management using write hints. In *Proc. of the USENIX File and Storage Technologies*, 2005.

[22] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezise, and Peter Camble. Sparse indexing: large scale, inline deduplication using sampling and locality. In *Proc. of the USENIX File and Storage Technologies*, February 2009.

[23] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[24] Nimrod Megiddo and D. S. Modha. Arc: A self-tuning, low overhead replacement cache. In *Proc. of USENIX File and Storage Technologies*, 2003.

[25] G. Milos, D. G. Murray, S. Hand, and M. Fetterman. Satori: Enlightened Page Sharing. In *Proc. of the Usenix Annual Technical Conference*, June 2009.

[26] Charles B. Morrey III and Dirk Grunwald. Peabody: The Time Travelling Disk. In *Proc. of the IEEE/NASA MSST*, 2003.

[27] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *Proc. of the ACM SOSP*, October 2001.

[28] Network Appliance, Inc. NetApp V-Series of Heterogeneous Storage Environments. http://media.netapp.com/documents/v-series.pdf.

[29] Cyril U. Orji and Jon A. Solworth. Doubly distorted mirrors. In *Proceedings of the ACM SIGMOD*, 1993.

[30] S. Quinlan and S. Dorward. Venti: A New Approach to Archival Storage. *Proc. of the USENIX File And Storage Technologies*, January 2002.

[31] Sean Rhea, Russ Cox, and Alex Pesterev. Fast, Inexpensive Content-Addressed Storage in Foundation. *Proc. of USENIX Annual Technical Conference*, June 2008.

[32] C. Ruemmler and J. Wilkes. Disk Shuffling. *Technical Report HPL-CSP-91-30, Hewlett-Packard Laboratories*, October 1991.

[33] Jon A. Solworth and Cyril U. Orji. Distorted Mirrors. *Proc. of PDIS*, 1991.

[34] Niraj Tolia, Michael Kozuch, Mahadev Satyanarayanan, Brad Karp, and Thomas Bressoud. Opportunistic use of content addressable storage for distributed file systems. *Proc. of the USENIX Annual Technical Conference*, 2003.

[35] Paul Vongsathorn and Scott D. Carson. A System for Adaptive Disk Rearrangement. *Softw. Pract. Exper.*, 20(3):225–242, 1990.

[36] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. *Proc. of USENIX OSDI*, 2002.

[37] C. K. Wong. Minimizing Expected Head Movement in One-Dimensional and Two-Dimensional Mass Storage Systems. *ACM Computing Surveys*, 12(2):167–178, 1980.

[38] Theodore M. Wong and John Wilkes. My Cache or Yours? Making Storage More Exclusive. In *Proc. of the USENIX Annual Technical Conference*, 2002.

[39] X. Yu, B. Gum, Y. Chen, R. Y. Wang, K. Li, A. Krishnamurthy, and T. E. Anderson. Trading Capacity for Performance in a Disk Array. *Proc. of USENIX OSDI*, 2000.

[40] C. Zhang, X. Yu, A. Krishnamurthy, and R. Y. Wang. Configuring and Scheduling an Eager-Writing Disk Array for a Transaction Processing Workload. In *Proc. of USENIX File and Storage Technologies*, January 2002.

[41] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. *Proc. of the USENIX File And Storage Technologies*, February 2008.

HydraFS: a High-Throughput File System for the HYDRAstor Content-Addressable Storage System

Cristian Ungureanu, Benjamin Atkin, Akshat Aranya, Salil Gokhale,
Stephen Rago, Grzegorz Całkowski, Cezary Dubnicki, and Aniruddha Bohra

NEC Laboratories America

cristian@nec-labs.com, atkin@google.com, aranya@nec-labs.com, salil@nec-labs.com,
sar@nec-labs.com, grzes@vmware.com, dubnicki@9livesdata.com, abohra@akamai.com

Abstract

A content-addressable storage (CAS) system is a valuable tool for building storage solutions, providing efficiency by automatically detecting and eliminating duplicate blocks; it can also be capable of high throughput, at least for streaming access. However, the absence of a standardized API is a barrier to the use of CAS for existing applications. Additionally, applications would have to deal with the unique characteristics of CAS, such as immutability of blocks and high latency of operations. An attractive alternative is to build a file system on top of CAS, since applications can use its interface without modification.

Mapping a file system onto a CAS system efficiently, so as to obtain high duplicate elimination and high throughput, requires a very different design than for a traditional disk subsystem. In this paper, we present the design, implementation, and evaluation of HydraFS, a file system built on top of HYDRAstor, a scalable, distributed, content-addressable block storage system. HydraFS provides high-performance reads and writes for streaming access, achieving 82–100% of the HYDRAstor throughput, while maintaining high duplicate elimination.

1 Introduction

Repositories that store large volumes of data are increasingly common today. This leads to high capital expenditure for hardware and high operating costs for power, administration, and management. A technique that offers one solution for increasing storage efficiency is data deduplication, in which redundant data blocks are identified, allowing the system to store only one copy and use pointers to the original block instead of creating redundant blocks. Deduplicating storage is ideally suited to backup applications, since they store similar data repeatedly, and with growing maturity is expected to become common in the data center for general application use.

Data deduplication can be achieved *in-line* or *off-line*. In both cases, data is eventually stored in an object store where objects are referenced through addresses derived from their contents. Objects can be entire files, blocks of data of fixed size, or blocks of data of variable size.

In a CAS system with in-line deduplication, the data blocks are written directly to the object store. Thus, they are not written to disk if they are deemed duplicates; instead, the address of the previously written block with the same contents is used. A CAS system with off-line deduplication first saves data to a traditional storage system, and deduplication processing is done later. This incurs extra I/O costs, as data has to be read and re-written, and requires additional storage space for keeping data in non-deduplicated form until the processing is complete.

While a CAS system with in-line deduplication does not have these costs, using it directly has two disadvantages: the applications have to be modified to use the CAS-specific API, and use it in such a way that the best performance can be obtained from the CAS system. To avoid the inconvenience of rewriting many applications, we can layer a file system on top of the object store. This has the advantage that it presents a standard interface to applications, permitting effective use of the CAS system to many applications without requiring changes. Additionally, the file system can mediate between the access patterns of the application and the ones best supported by the CAS system.

Designing a file system for a distributed CAS system is challenging, mainly because blocks are immutable, and the I/O operations have high latency and jitter. Since blocks are immutable, all data structures that hold references to a block must be updated to refer to the new

address of the block whenever it is modified, leading to multiple I/O operations, which is inefficient. Distributed CAS systems also impose high latencies in the I/O path, because many operations must be done in the critical path.

While file systems have previously been built for CAS systems, most have scant public information about their design. One notable exception is LBFS [18], which focuses on nodes connected by low bandwidth, wide area networks. Because of the low bandwidth, it is not targeted at high-throughput applications, and poses different challenges for the file system designers.

This paper describes the design, implementation, and evaluation of HydraFS, a file system layered on top of a distributed, content-addressable back end, HYDRAstor [5] (also called Hydra, or simply the "block store"). Hydra is a multi-node, content-addressable storage system that stores blocks at configurable redundancy levels and supports high-throughput reads and writes for streams of large blocks. Hydra is designed to provide a content-addressable block device interface that hides the details of data distribution and organization, addition and removal of nodes, and handling of disk and node failures.

HydraFS was designed for high-bandwidth streaming workloads, because its first commercial application is as part of a backup appliance. The combination of CAS block immutability, high latency of I/O operations, and high bandwidth requirements brings forth novel challenges for the architecture, design, and implementation of the file system. To the best of our knowledge, HydraFS is the first file system built on top of a distributed CAS system that supports high sequential read and write throughput while maintaining high duplicate elimination.

We faced three main challenges in achieving high throughput with HydraFS. First, updates are more expensive in a CAS system, as all metadata blocks that refer to a modified block must be updated. This metadata comprises mappings between an inode number and the inode data structure, the inode itself, which contains file attributes, and file index blocks (for finding data in large files). Second, cache misses for metadata blocks have a significant impact on performance. Third, the combination of high latency and high throughput requires a large write buffer and read cache. At the same time, if these data structures are allowed to grow without bound, the system will thrash.

We overcome these challenges through three design strategies. First, we decouple data and metadata processing through the use of a log [10]. This split allows the metadata modifications to be batched and applied efficiently. We describe a metadata update technique that maintains consistency without expensive locking. Second, we use fixed-size caches and use admission control to limit the number of concurrent file system operations such that their processing needs do not exceed the available resources. Third, we introduce a second-order cache to reduce the number of misses for metadata blocks. This cache also helps reduce the number of operations that are performed in the context of a read request, thus reducing the response time.

Our experimental evaluation confirms that HydraFS enables high-throughput sequential reads and writes of large files. In particular, HydraFS is able to support sequential writes to a single file at 82–100% of the underlying Hydra storage system's throughput. Although HydraFS is optimized for high-throughput streaming file access, its performance is good enough for directory operations and random file accesses, making it feasible for bulk data transfer applications to use HydraFS as a general-purpose file system for workloads that are not metadata-intensive.

This paper makes the following contributions. First, we present a description of the challenges in building a file system on top of a distributed CAS system. Second, we present the design of a file system, HydraFS, that overcomes these challenges, focusing on several key techniques. Third, we present an evaluation of the system that demonstrates the effectiveness of these techniques.

2 Hydra Characteristics

HydraFS acts as a front end for the Hydra distributed, content-addressable block store (Figure 1). In this section, we present the characteristics of Hydra and describe the key challenges faced when using it for applications, such as HydraFS, that require high throughput.

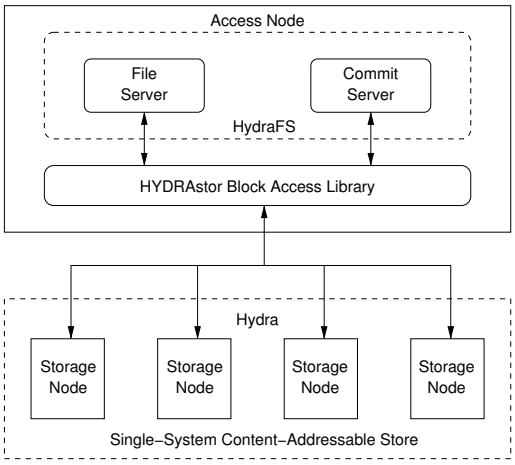


Figure 1: HYDRAstor Architecture.

2.1 Model

HydraFS runs on an *access node* and communicates with the block store using a library that hides the distributed

nature of the block store. Even though the block store is implemented across multiple *storage nodes*, the API gives the impression of a single system.

The HYDRAstor block access library presents a simple interface:

Block write The caller provides a block to be written and receives in return a receipt, called the *content address*, for the block. If the system can determine that a block with identical content is already stored, it can return its content address instead of generating a new one, thus eliminating duplicated data. Multiple resiliency levels are available to control the amount of redundant information stored, thereby allowing control over the number of component failures that a block can withstand.

The block access library used on the access node has the option of querying the storage nodes for the existence of a block with the same hash key to avoid sending the block over the network if it already exists. This is a block store feature, imposing only a slight increase in the latency of the write operations, that is already tolerated by the file system design.

Block read The caller provides the content address and receives the data for the block in return.

Searchable block write Given a block and a *label*, the block is stored and associated with the label. If a block with the same label but different content is already stored, the operation fails. Labels can be any binary data chosen by the client, and need not be derived from the contents of the block.

Two types of searchable blocks are supported: *retention roots* that cause the retention of all blocks reachable from them, and *deletion roots* that mark for deletion the retention roots with the same labels. Periodically, a garbage collection process reclaims all blocks that are unreachable from retention roots not marked for deletion.

Searchable block read Given a label, the contents of the associated retention root are returned.

The searchable block mechanism provides a way for the storage system to be self-contained. In the absence of a mechanism to retrieve blocks other than through their content address, an application would have to store at least one content address *outside* the system, which is undesirable.

2.2 Content Addresses

In HYDRAstor, content addresses are opaque to clients (in this case, the filesystem). The block store is responsible for calculating a block's content address based on a secure, one-way hash of the block's contents and other information that can be used to retrieve the block quickly.

For the same data contents, the block store can return the same content address, although it is not obliged to do so. For example, a block written at a higher resiliency level would result in a different content address even if an identical block were previously written at a lower resiliency level. The design also allows for a byte-by-byte comparison of newly-written data blocks whose hashes match existing blocks. Collisions (different block contents hashing to the same value) would be handled by generating a different content address for each block. For performance reasons, and given that the hash function is strong enough to make collisions statistically unlikely, the default is to not perform the check.

Because the content address contains information that the file system does not have, it is impossible for the file system to determine the content address of a block in advance of submitting it to the block store. At first blush, since the latency of the writes is high, this might seem like a problem for performance, because it reduces the potential parallelism of writing blocks that contain pointers to other blocks. However, this is not a problem for two reasons. First, even if we were to write all blocks in parallel, we still would have to wait for all child blocks to be persistent before writing the searchable retention root. The interface is asynchronous: write requests can complete in a different order than that in which they were submitted. If we were to write the searchable block without waiting for the children and the system were to crash, the file system would be inconsistent if the retention root made it to disk before all of its children.

Second, the foreground processing, which has the greatest effect on write performance, writes only shallow trees of blocks; the trees of higher depth are written in the background, so the reduction in concurrency is not significant enough to hurt the performance of streaming writes. Thus, although the high latency of operations is a challenge for attaining good performance, the inability of the file system to calculate content addresses on its own does not present an additional problem.

2.3 Challenges

Hydra presents several challenges to implementing a file system that are not encountered with conventional disk subsystems. Some of the most notable are: *(i)* blocks are immutable, *(ii)* the latency of the block operations is very high, and *(iii)* a *chunking* algorithm must be used to determine the block boundaries that maximize deduplication, and this results in blocks of unpredictable and varied sizes.

2.3.1 Immutable Blocks

When a file system for a conventional disk subsystem needs to update a block, the file system can simply rewrite it, since the block's address is fixed. The new contents become visible without requiring further writes, regardless of how many metadata blocks need to be traversed to reach it.

A CAS system, however, has to store a new block, which may result in the new block's address differing from the old block's address. Because we are no longer interested in the contents of the old block, we will informally call this an "update." (Blocks that are no longer needed are garbage collected by Hydra.) But to reach the new block, we need to update its parent, and so on recursively up to the root of the file system. This leads to two fundamental constraints on data structures stored in a CAS system.

First, because the address of a block is derived from a secure, one-way hash of the block's contents, it is impossible for the file system to store references to blocks not yet written. Since blocks can only contain pointers to blocks written in the past, and more than one block can contain the same block address, the blocks form directed acyclic graphs (DAGs).

Second, the height of the DAG should be minimized to reduce the overhead of modifying blocks. The cost to modify a block in a file system based on a conventional disk subsystem is limited to the cost to read and write the block. In a CAS-based file system, however, the cost to modify a block also includes the cost to modify the chain of blocks that point to the original block. While this problem also occurs in no-overwrite file systems, such as WAFL [11], it is exacerbated by higher Hydra latencies, as discussed in the next section.

2.3.2 High Latency

Another major challenge that Hydra poses is higher latencies than conventional disk subsystems. In a conventional disk subsystem, the primary task in reading or writing a disk block is transferring the data. In Hydra however, much more work must be done before an I/O operation can be completed. This includes scanning the entire contents of the block to compute its content address, compressing or uncompressing the block, determining the location where the block is (or will be) stored, fragmenting or reassembling the blocks that are made up of smaller fragments using error-correcting codes, and routing these fragments to or from the nodes where they reside. While conventional disk subsystems have latencies on the order of milliseconds to tens of milliseconds, Hydra has latencies on the order of hundreds of milliseconds to seconds.

An even higher contributor to the increased latency comes from the requirement to support high-throughput reads. With conventional disk subsystems, placing data in adjacent blocks typically ensures high-throughput reads. The file system can do that because there is a clear indication of adjacency: the block number. However, a CAS system places data based on the block content's hash, which is unpredictable. If Hydra simply places data contiguously based on temporal affinity, as the number of streams written concurrently increases, the blocks of any one stream are further and further apart, reducing the locality and thus causing low read performance.

To mitigate this problem, the block store API allows the caller to specify a *stream hint* for every block write. The block store will attempt to co-locate blocks with the same stream hint by delaying the writes until a sufficiently large number of blocks arrive with the same hint. The decision of what blocks should be co-located is up to the file system; in HydraFS all blocks belonging to the same file are written with the same hint.

The write delay necessary to achieve good read performance depends by the number of concurrent write streams. The default value of the delay is about one second, which is sufficient for supporting up to a hundred concurrent streams. Thus, the write latency is sacrificed for the sake of increased read performance. To cope with the large latencies but still deliver high throughput, the file system must be able to issue a large number of requests concurrently.

2.3.3 Variable Block Sizes

The file system affects the degree of deduplication by how it divides files into blocks, a process we call *chunking*. Studies have shown that variable-size chunking provides better deduplication than fixed-size chunking ([15], [20]). Although fixed-size chunking can be sufficient for some applications, backup streams often contain duplicate data, possibly shifted in the stream by additions, removals, or modifications of files.

Consider the case of inserting a few bytes into a file containing duplicate contents, thereby shifting the contents of the rest of the file. If fixed-size chunking is used, and the number of bytes is not equal to the chunk size, duplicate elimination would be defeated for the range of file contents from the point of insertion through the end of the file. Instead, we use a content-defined chunking algorithm, similar to the one in [18], that produces chunks of variable size between a given minimum and maximum.

This design choice affects the representation of files. With a variable block size, an offset into a file cannot be mapped to the corresponding block by a simple mathematical calculation. This, along with the desire to have DAGs of small height, led us to use balanced tree structures.

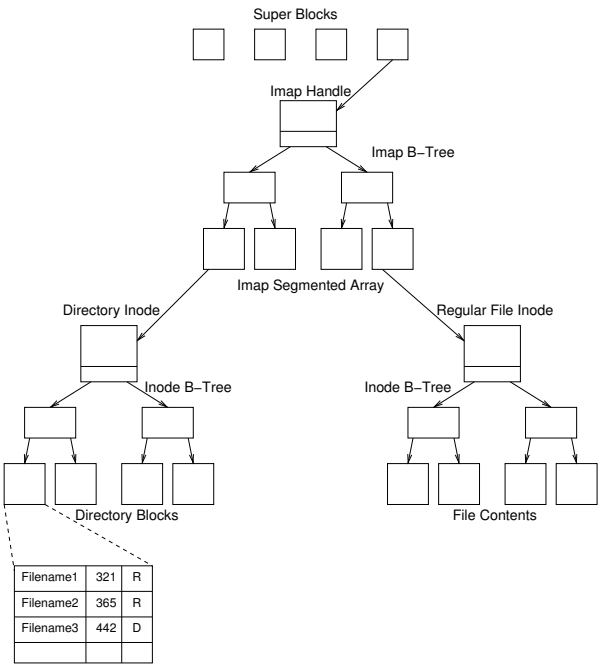


Figure 2: HydraFS persistent layout.

3 File System Design

HydraFS design is governed by four key principles. First, the primary concern is the high throughput of sequential reads and writes. Other operations, such as metadata operations, file overwrites, and simultaneous reads and writes to the same file, are supported, but are not the target for optimization. Second, because of the high latencies of the block store, the number of dependent I/O operations must be minimized. At the same time, the system must be highly concurrent to obtain high throughput. Third, the data availability guarantees of HydraFS must be no worse than those of the standard Unix file systems. That is, while data acknowledged before an `fsync` may be lost in case of system crash, once an `fsync` is acknowledged to the application, the data must be persistent. Fourth, the file system must efficiently support both local file system access and remote access over NFS and CIFS.

3.1 File System Layout

Figure 2 shows a simplified view of the HydraFS file system block tree. The file system layout is structured as a DAG, with the root of the structure stored in a searchable block. The searchable block contains the file system super block, which holds the address of the *inode map* (called the "imap") together with the current file system version number and some statistics. The imap is conceptually similar to the inode map used in the Log-Structured File System [23]. In HydraFS, the imap is a variable-length array of content addresses and allocation status, stored as a B-tree. It is used to translate inode numbers into inodes, as well as to allocate and free inode numbers.

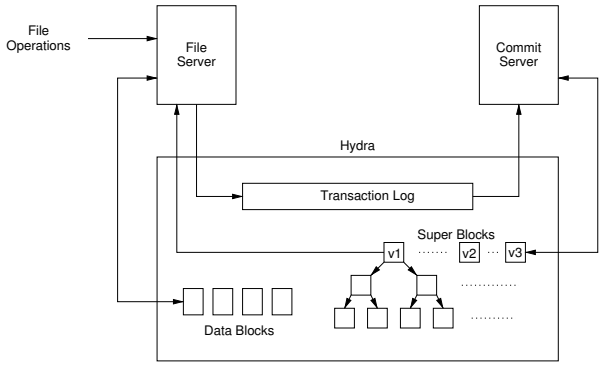


Figure 3: HydraFS Software Architecture.

A regular file inode indexes data blocks with a B-tree so as to accommodate very large files [27] with variable-size blocks. Regular file data is split up into variable-size blocks using a chunking algorithm that is designed to increase the likelihood that the same data written to the block store will generate a match. Thus, if a file is written to the block store on one file system, and then written to another file system using the same block store, the only additional blocks that will be stored by the block store will be the metadata needed to represent the new inode, and its DAG ancestors: the imap and the superblock. The modifications of the last two are potentially amortized over many inode modifications.

Although the immutable nature of Hydra's blocks naturally allows for filesystem snapshots, this feature is not yet exposed to the applications that use HydraFS.

3.2 HydraFS Software Architecture

HydraFS is implemented as a pair of user-level processes that cooperate to provide file system functionality (see Figure 3). The FUSE file system module [8] provides the necessary glue to connect the servers to the Linux kernel file system framework. The *file server* is responsible for managing the file system interface for clients; it handles client requests, records file modifications in a persistent transaction log stored in the block store, and maintains an in-memory cache of recent file modifications. The *commit server* reads the transaction log from the block store, updates the file system metadata, and periodically generates a new file system version.

This separation of functionality has several advantages. First, it simplifies the locking of file system metadata (discussed further in Section 3.3). Second, it allows the

commit server to amortize the cost of updating the file system's metadata by batching updates to the DAG. Third, the split allows the servers to employ different caching strategies without conflicting with each other.

3.3 Write Processing

When an application writes data to a file, the file server accumulates the data in a buffer associated with the file's inode and applies a content-defined chunking algorithm to it. When chunking finds a block boundary, the data in the buffer up to that point is used to generate a new block. The remaining data is left in the buffer to form part of the next block. There is a global limit on the amount of data that is buffered on behalf of inodes, but not yet turned into blocks, to prevent the memory consumption of the inode buffers from growing without bound. When the limit is reached, some buffers are flushed, their content written to Hydra even though the chunk boundaries are no longer content-defined.

Each new block generated by chunking is marked dirty and immediately written to Hydra. It must be retained in memory until Hydra confirms the write. The file server must have it available in case the write is followed by a read of that data, or in case Hydra rejects the block write due to an overloaded condition (the operation is resubmitted after a short delay). When Hydra confirms the write, the block is freed, but its content address is added to the *uncommitted block table* with a timestamp and the byte range that corresponds to the block.

The uncommitted block table is a data structure used for keeping modified file system metadata in memory. Since there is no persistent metadata block pointing to the newly-written data block, this block is not yet reachable in a persistent copy of the file system.

An alternative is to update the persistent metadata immediately, but this has two big problems. The first is that each data block requires the modification of all metadata blocks up to the root. This includes inode index blocks, inode attribute block, and imap blocks. Updating all of them for every data block modification creates substantial I/O overhead. The second is that the modification to these data structures would have to be synchronized with other concurrent operations performed by the file server. Since the metadata tree can only be updated one level at a time (a parent can be written only after the writes of all children complete), propagation up to the root has a very high latency. Locking the imap for the duration of these writes would reduce concurrency considerably, resulting in extremely poor performance. Thus, we chose to keep dirty metadata structures in memory and delegate the writing of metadata to the commit server.

When the commit server finally creates a new file system super block, the file server can clean its dirty metadata structures (see Section 3.4). To provide persistence guarantees, the metadata operations are written to a log which is kept persistently in Hydra until they are executed by the commit server.

Sequentially appending data to files exhibits the best performance in HydraFS. Random writes in HydraFS incur more overhead than appends because of the chunking process that decides the boundaries of the blocks written to Hydra. The boundaries depend on the content of the current write operation, but also on the file data adjacent to the current write range (if any). Thus, a random write to the file system might generate block writes to the block store that include parts of blocks already written, as well as any data that was buffered but not yet written since it was not a complete chunk.

3.4 Metadata Cleaning

The file server must retain dirty metadata as a consequence of delegating metadata processing to the commit server to avoid locking. This data can only be discarded once it can be retrieved from the block store. For this to happen, the commit server must sequentially apply the operations it retrieves from the log written by the file server, create a new file system DAG, and commit it to Hydra.

To avoid unpredictable delays, the commit server generates a new file system version periodically, allowing the file server to clean its dirty metadata proactively. Instead, if the file server waits until its cache fills up before asking the commit server to generate a new root, then the file server would stall until the commit server completes writing all the modified metadata blocks. As mentioned before, this can take a long time, because of the sequential nature of writing content-addressable blocks along a DAG path.

Metadata objects form a tree that is structurally similar to the block tree introduced in Section 3.1. To simplify metadata cleaning, the file server does not directly modify the metadata objects as they are represented on Hydra. Instead, all metadata modifications are maintained in separate lookup structures, with each modification tagged with its creation time. With this approach, the metadata that was read from Hydra is always clean and can be dropped from the cache at any time, if required.

When the file server sees that a new super block has been created, it can clean the metadata objects in a top-down manner. Cleaning a metadata object involves replacing its cached clean state (on-Hydra state) with a new version, and dropping all metadata modification records that have been incorporated into the new version.

The top-down restriction is needed to ensure that a discarded object will not be re-fetched from Hydra using an out-of-date content address. For example, if the file server were to drop a modified inode before updating the imap

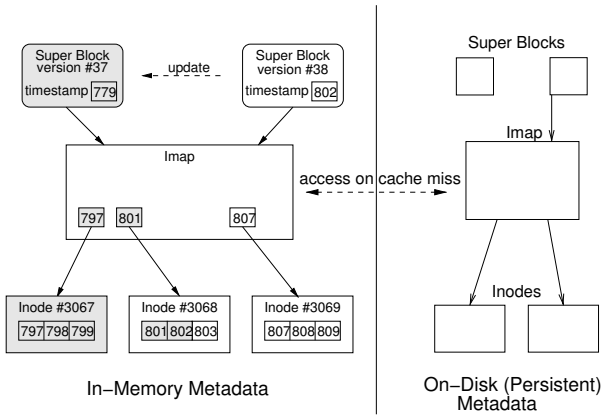


Figure 4: Cleaning of In-Memory Metadata.

first, the imap would still refer to the old content address and a stale inode would be fetched if the inode were accessed again.

Figure 4 shows an example of metadata cleaning. The file server keeps an in-memory list of inode creation (or deletion) records that modify the imap, as well as uncommitted block table records for the inodes, consisting of content addresses with creation timestamps (and offset range, not shown). The file server might also have cached blocks belonging to the old file system version (not shown). Inode 3067 can be discarded, because all of its modifications are included in the latest version of the super block. Inode 3068 cannot be removed, but it can be partially cleaned by dropping content addresses with timestamps 801 and 802. Similarly, creation records up to timestamp 802 can be dropped from the imap. Note that in-memory inodes take precedence over imap entries; the stale imap information for inode 3068 will not be used as long as the inode stays in memory.

3.5 Admission Control

Both servers accept new events for processing after first passing the events through admission control, a mechanism designed to limit the amount of memory consumed. Limits are determined by the amount of memory configured for particular objects, such as disk blocks and inodes. When an event arrives, the worst-case needed allocation size is reserved for each object that might be needed to process the event. If memory is available, the event is allowed into the server's set of active events for processing. Otherwise, the event blocks.

During its lifetime, an event can allocate and free memory as necessary, but the total allocation cannot exceed the reservation. When the event completes, it relinquishes the reservation, but it might not have freed all the memory it allocated. For example, a read event leaves blocks in the cache. Exhaustion of the memory pool triggers a reclamation function that frees cached objects that are clean.

Admission control solves two problems. First, it limits the amount of active events, which in turn limits the amount of heap memory used. This relieves us from having to deal with memory allocation failures, which can be difficult to handle, especially in an asynchronous system where events are in various stages of completion. Second, when the memory allocated for file system objects is tuned with the amount of physical memory in mind, it can prevent paging to the swap device, which would reduce performance.

3.6 Read Processing

The file system cannot respond to a read request until the data is available, making it harder to hide high CAS latencies. To avoid I/O in the critical path of a read request, HydraFS uses aggressive read-ahead for sequential reads into an in-memory, LRU cache, indexed by content address. The amount of additional data to be read is configurable with a default of 20MB.

To obtain the content addresses of the data blocks that cover the read-ahead range, the metadata blocks that store these addresses must also be read from the inode's B-tree. This may require multiple reads to fetch all blocks along the paths from the root of the tree to the leaf nodes of interest. To amortize the I/O cost, HydraFS caches both metadata blocks and the data blocks, uses large leaf nodes, and high fan-out for internal nodes.

Unfortunately, the access patterns for data and metadata blocks differ significantly. For sequential reads, accesses to data blocks are close together, making LRU efficient. In contrast, because of the large fan-out, consecutive metadata block accesses might be separated by many accesses to data blocks, making metadata eviction more likely. An alternative is to use a separate cache for data and metadata blocks, but this does not work well in cases when the ratio of data to metadata blocks differs from the ratio of the two caches. Instead, we use a single *weighted LRU* cache, where metadata blocks have a higher weight, making them harder to evict.

To further reduce the overhead of translating offset-length ranges to content addresses, we use a per-inode look-aside buffer, called the *fast range map* (FRM), that maintains a mapping from an offset range to the content address of the block covering it. The FRM has a fixed size, is populated when a range is first translated, and is cleared when the corresponding inode is updated.

Finally, we also introduce a read-ahead mechanism for metadata blocks to eliminate reads in the critical path of the first access to these blocks. The B-tree read-ahead augments the priming of the FRM for entries that are likely to be needed soon.

3.7 Deletion

When a file is deleted in HydraFS, that file is removed from the current version of the file system namespace. Its storage space, however, remains allocated in the block store until no more references to its blocks exist and the back end runs its garbage collection cycle to reclaim unused blocks. The garbage collection is run as an administrative procedure that requires all modified cached data to be flushed by HydraFS to Hydra to make sure that there are no pointers to blocks that might be reclaimed.

Additional references to a file's blocks can come from two sources: other files that contain the same chunk of data, and older versions of the file system that contain references to the same file. References need not originate from the same file system, however. Since all file systems share the same block store, blocks can match duplicates from other file systems.

When a new version of a file system is created, the oldest version is marked for deletion by writing a deletion root corresponding to its retention root. The file system only specifies which super blocks are to be retained and which are to be deleted, and Hydra manages the reference counts to decide which blocks are to be retained and which are to be freed.

The number of file system versions retained is configurable. These versions are not currently exposed to users; they are retained only to provide insurance should a file system need to be recovered.

Active log blocks are written as shallow trees headed by searchable blocks. Log blocks are marked for deletion as soon as their changes are incorporated into an active version of the file system.

4 Evaluation

HydraFS has been designed to handle sequential workloads operating under unique constraints imposed by the distributed, content-addressable block store. In this section, we present evidence that HydraFS supports high throughput for these workloads while retaining the benefits of block-level duplicate elimination. We first characterize our block storage system, focusing on issues that make it difficult to design a file system on top of it. We then study HydraFS behavior under different workloads.

4.1 Experimental Setup

All experiments were run on a setup of five computers similar to Figure 1. We used a 4-server configuration of storage nodes, in which each server had two dual-core, 64-bit, 3.0 GHz Intel Xeon processors, 6GB of memory, and six 7200 RPM MAXTOR 7H500F0 SATA disks, of which five were used to store blocks, and one was used for logging by Hydra. Its redundancy is given by an erasure coding scheme [1] using 9 original and 3 redundant fragments. A similar hardware configuration was used for the file server, but with 8GB of memory and an ext3 file system on a logical volume split across two 15K RPM Fujitsu MAX3073RC SAS disks using hardware RAID: this file system was used for logging in HydraFS experiments, and for storing data in ext3 experiments. All servers run a 2.6.9 Linux kernel, because this was the version that was used in the initial product release. (It has since been upgraded to a more recent version; regardless, the only local disk I/O on the access node is for logging, so improvements in the disk I/O subsystem won't affect our performance appreciably.)

4.2 HydraFS Efficiency

The goal in this section is to characterize the efficiency of HydraFS and to demonstrate that it comes close to the performance supported by our block store. Unfortunately, since Hydra exports a non-standard API and HydraFS is designed for this API, it is not possible for us to use a common block store for both HydraFS and a disk-based file system, such as ext3. *It is important to note that we are not interested in the absolute performance of the two file systems, but how much the performance degrades when using a file system compared to a raw block device.*

To compare the efficiencies of HydraFS and ext3, we used identical hardware, configured as follows. We exported an ensemble of 6 disks on each storage node as an iSCSI target using a software RAID5 configuration with one parity disk. We used one access node as the iSCSI initiator and used software RAID0 to construct an ensemble that exposes one device node. We used a block size of 64KB for the block device and placed an ext3 file system on it. The file system was mounted with the `noatime` and `nodiratime` mount options. This configuration allows ext3 access to hardware resources similar to Hydra, although its resilience was lower than that of Hydra, as it does not protect against node failure or more than one disk failure per node.

Sequential Throughput: In this experiment, we use a synthetic benchmark to generate a workload for both the HydraFS and ext3 file systems. This benchmark generates a stream of reads or writes with a configurable I/O size using blocking system calls and issues a new request as soon as the previous request completes. Additionally, this benchmark generates data with a configurable fraction of duplicate data, which allows us to study the behavior of HydraFS with variable data characteristics. The throughput of the block store is measured with an application that uses the CAS API to issue in parallel as many block operations as accepted by Hydra, thus exhibiting the maximum level of concurrency possible.

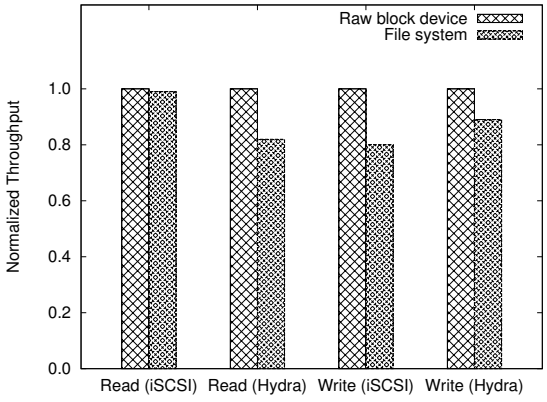


Figure 5: Comparison of raw device and file system throughput for iSCSI and Hydra

Figure 5 shows the read and write throughput achieved by ext3 and HydraFS against the raw block device throughput of the iSCSI ensemble and Hydra respectively. We observe that while the read throughput of ext3 is comparable to that of its raw device, HydraFS read throughput is around 82% of the Hydra throughput. For the write experiment, while ext3 throughput degrades to around 80% of the raw device, HydraFS achieves 88% of Hydra throughput, in spite of the block store's high latency.

Therefore, we conclude that the HydraFS implementation is efficient and the benefits of flexibility and generality of the file system interface do not lead to a significant loss of performance. The performance difference comes mostly from limitations on concurrency imposed by dependencies between blocks, as well as by memory management in HydraFS, which do not exist in raw Hydra access.

Metadata Intensive Workloads: To measure the performance of our system with a metadata-intensive workload, we used Postmark [12] configured with an initial set of 50,000 files in 10 directories, with file sizes between 512B and 16KB. We execute 30,000 transactions for each run of the benchmark. Postmark creates a set of files, followed by a series of transactions involving read or write followed by a create or delete. At the end of the run, the benchmark deletes the entire file set.

Table 1 shows the file creation and deletion rate with and without transactions, including the overall rate of transactions for the experiment. A higher number of transactions indicates better performance for metadata-intensive workloads.

We observe that the performance of HydraFS is much lower than that of ext3. Creating small files presents the worst case for Hydra, as the synchronous metadata operations are amortized over far fewer reads and writes than with large files. Moreover, creation and deletion are limited by the number of inodes HydraFS creates without going through the metadata update in the commit server. We keep this number deliberately low to ensure that the system does not accumulate a large number of uncommitted blocks that increase the turnaround times for the commit server processing, increasing unpredictably the latency of user operations. In contrast, ext3 has no such limitations and all metadata updates are written to the journal.

	Create		Delete		Overall
	Alone	Tx	Alone	Tx	
ext3	1,851	68	1,787	68	136
HydraFS	61	28	676	28	57

Table 1: Postmark comparing HydraFS with ext3 on similar hardware

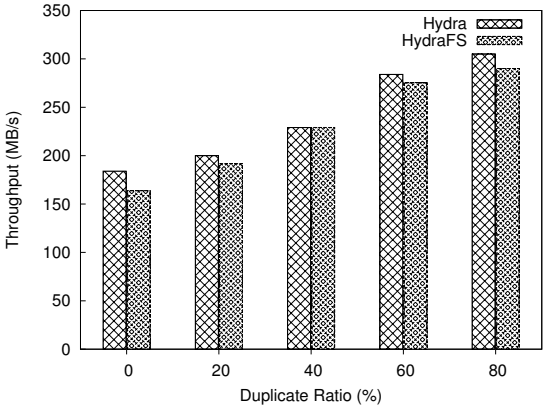


Figure 6: Hydra and HydraFS write throughput with varying duplicate ratio

4.3 Write Performance

In the experiment, we write a 32GB file sequentially to HydraFS using a synthetic benchmark. The benchmark uses the standard file system API for the HydraFS experiment and uses the custom CAS API for the Hydra experiment.

We vary the ratio of duplicate data in the write stream and report the throughput. For repeatability in the presence of variable block sizes and content-defined chunking, our benchmark is designed to generate a configurable average block size, which we set to 64KB in all our experiments.

Figure 6 shows the write throughput when varying the fraction of duplicates in the write stream from no duplicates (0%) to 80% in increments of 20%. We make two observations from our results. First, the throughput increases linearly as the duplicate ratio increases. This is

as expected for duplicate data as the number of I/Os to disk is correspondingly reduced. Second, for all cases, the HydraFS throughput is within 12% of the Hydra throughput. Therefore, we conclude that HydraFS meets the desired goal of maintaining high throughput.

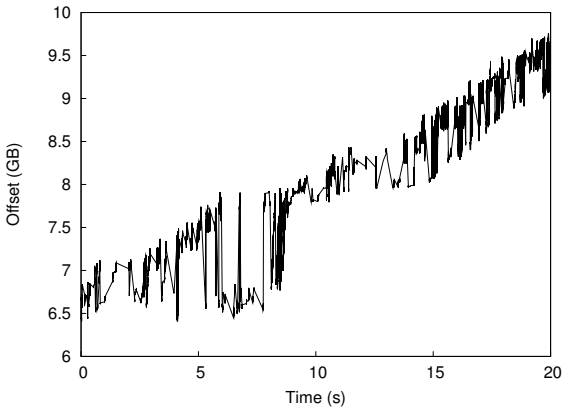


Figure 7: Write completion order

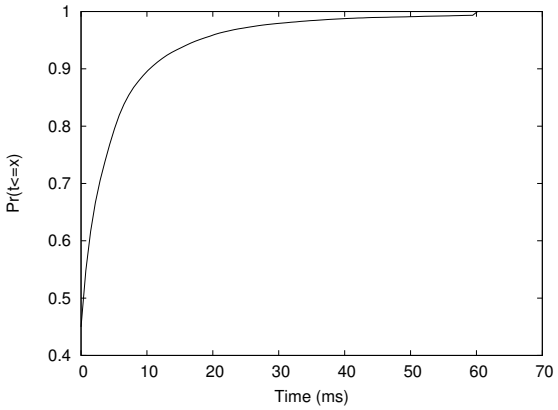


Figure 8: Write event lifetimes

To support high-throughput streaming writes, HydraFS uses a write-behind strategy and does not perform any I/O in the critical path. To manage its memory resources and to prevent thrashing, HydraFS uses a fixed size write buffer and admission control to block write operations before they consume any resources.

Write Behind: Figure 7 shows the order of I/O completions as they arrive from Hydra during a 20-second window of execution of the sequential write benchmark. In an ideal system, the order of completion would be the same as the order of submission and the curve shown in the figure would be a straight line. We observe that in the worst case the gap between two consecutive block completions in this experiment can be as large as 1.5GB, a testament to the high jitter exhibited by Hydra. Consequently, the latency of an internal compound operation requiring many block writes to the back end will experience a latency higher than the average even if all blocks are written in parallel.

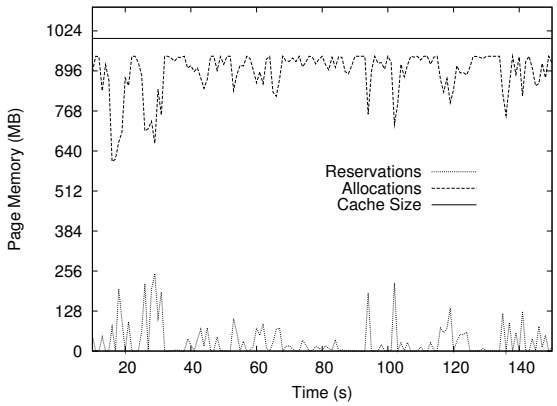


Figure 9: Resource reservation and allocations

To further understand the write behavior, we show the Cumulative Distribution Function (CDF) of the write event lifetimes in the system in Figure 8. The write event is created when the write request arrives at HydraFS and is destroyed when the response is sent to the client. Figure 8 shows that the 90^{th} percentile of write requests take less than 10 ms.

Admission control: In the experiments above, we show that HydraFS is highly concurrent even when the underlying block store is bursty and has high latency. To prevent the system from swapping under these conditions, we use admission control (see Section 3.5). In an ideal system, the allocations must be close to the size of the write buffer and the unused resources must be small to avoid wasting memory. Figure 9 shows the reservations and allocations in the system during a streaming write test. We observe that with admission control, HydraFS is able to maintain high memory utilization and only a fraction of the reserved resources are unused.

Commit Server Processing: Commit server processing overheads are much lower than file server overheads and we observe its CPU utilization to be less than 5% of the file server's utilization for all the experiments above. This allows the commit server to generate new versions well in advance of the file server filling up with dirty metadata, thus avoiding costly file server stalls.

4.4 Read Ahead Caching

In the following experiments, we generate a synthetic workload where a client issues sequential reads for 64KB blocks in a 32GB file. All experiments were performed with a cold cache and the file system was unmounted be-

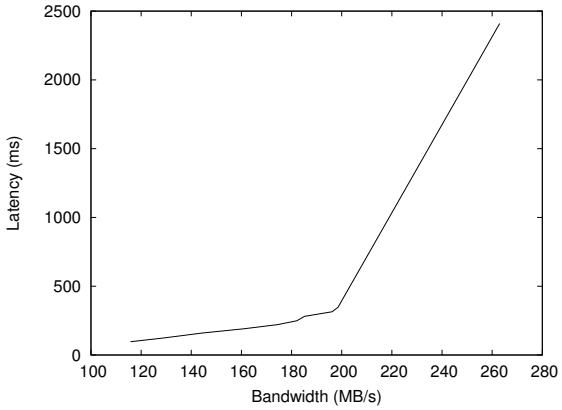


Figure 10: Read throughput vs. average latency

tween runs. Unless otherwise specified, the read-ahead window is fixed at 20MB.

To characterize the read behavior, we study how the read latency varies at different throughput levels. Hydra responds immediately to read requests when data is available. In this experiment we vary the offered load to Hydra by limiting the number of outstanding read requests and measure the time between submitting the request and receiving the response. We limit the number of outstanding read requests by changing the read ahead window from 20MB to 140MB in increments of 15MB.

Figure 10 shows the variation of average latency of read requests when the Hydra throughput is varied. From the figure, we observe that the read latency at low throughput is around 115 ms and increases linearly until the throughput reaches 200MB/s. At higher throughput levels, the latency increases significantly. These results show that the read latencies with Hydra are much higher than other block store latencies. This implies that aggressive read-ahead is essential to maintain high read throughput.

Optimizations: As described in Section 3, to maintain high throughput, we introduced two improvements - Fast Range Map and B-tree Read Ahead (BTreeRA). For a sequential access pattern, once data blocks are read, they are not accessed again. However, the metadata blocks (B-tree blocks) are accessed multiple times, often with a large inter-access gap. Both our optimizations, FRM and BTreeRA, target the misses of metadata blocks.

Table 2 shows the evolution of the read performance with introduction of these mechanisms. The FRM optimization reduces multiple accesses to the metadata blocks leading to a 23% improvement in throughput. BTreeRA reduces cache misses for metadata blocks by issuing read ahead for successive spines of the B-tree concurrently with collecting index data from one spine. Without this prefetch, the nodes populating the spine of the B-tree must be fetched when initiating a read. Moreover, the address

	Thrpt (MB/s)	Accesses	Misses	
			Data	Metadata
Base	134.3	486,966	1,577	1,011
FRM	166.1	210,480	871	1,593
FRM+BTreeRA	183.2	211,632	438	945

Table 2: Effect of read path optimizations

of the block at the next level is available only after the current block is read from Hydra. For large files, with multiple levels in the tree, this introduces a significant latency, which would cause a read stall.

To confirm the hypothesis that the throughput improvements are from reduced metadata accesses and cache misses, Table 2 also shows the number of accesses and the number of misses in the cache for all three cases. We make the following observations: first, our assumption that improving the metadata miss rate has significant impact on read throughput is confirmed. Second, our optimizations add a small memory and CPU overhead but can improve the read throughput by up to 36%.

5 Related Work

Several existing systems use content-addressable storage to support enterprise applications. Venti [21] uses fixed-size blocks and provides archival snapshots of a file system, but since it never deletes blocks, snapshots are made at a low frequency to avoid overloading the storage system with short-lived files. In contrast, HydraFS uses variable-size blocks to improve duplicate elimination and creates file system snapshots more frequently, deleting the oldest version when a new snapshot is created; this is enabled by Hydra providing garbage collection of unreferenced blocks.

Centera [6] uses a cluster of storage nodes to provide expandable, self-managing archival storage for immutable data records. It provides a file system interface to the block store through the Centera Universal Access (CUA), which is similar to the way an access node exports HydraFS file systems in HYDRAstor. The main difference is that the entire HydraFS file system image is managed in-line by storing metadata in the block store as needed; the CUA keeps its metadata locally and makes periodic backups of it to the block store in the background.

Data Domain [4, 31] is an in-line deduplicated storage system for high-throughput backup. Like HydraFS, it uses variable-size chunking. An important difference is that their block store is provided by a single node with RAID-ed storage, whereas Hydra is composed of a set of nodes, and uses erasure coding for configurable resilience at the individual block level.

Deep Store [29] is an architecture for archiving immutable objects that can be indexed by searchable metadata tags. It uses variable-size, content-defined chunks combined with delta compression to improve duplicate elimination. A simple API allows objects to be stored and retrieved, but no attempt is made to make objects accessible through a conventional file system interface.

Many file system designs have addressed providing high performance, fault-tolerant storage for clients on a local area network. The Log-Structured File System (LFS) [23] and Write-Anywhere File Layout (WAFL) [11] make use of specialized file system layouts to allow a file server to buffer large volumes of updates and commit them to disk sequentially. WAFL also supports snapshots that allow previous file system versions to be accessed. LFS uses an imap structure to cope with the fact that block addresses change on every write. WAFL uses an "inode file" containing all the inodes, and updates the relevant block when an inode is modified; HydraFS inodes might contain data and a large number of pointers, so they are stored in separate blocks. Neither LFS nor WAFL support in-line duplicate elimination. Elephant [24] creates new versions of files on every modification and automatically selects "landmark versions," incorporating major changes, for long-term retention. The Low-Bandwidth File System [18] makes use of Rabin fingerprinting [16, 22] to identify common blocks that are stored by a file system client and server, to reduce the amount of data that must be transferred over a low-bandwidth link between the two when the client fetches or updates a file.

The technique of building data structures using hash trees [17] has been used in a number of file systems. SFSRO [7] uses hash trees in building a secure read-only file system. Venti [21] adds duplicate elimination to make a content-addressable block store for archival storage, which can be used to store periodic snapshots of a regular file system. Ivy [19] and OceanStore [14] build on top of wide-area content-addressable storage [26, 30]. While HydraFS is specialized for local-area network performance, Ivy focuses on file system integrity in a multi-user system with untrusted participants, and OceanStore aims to provide robust and secure wide-area file access. Pastiche [3] uses content hashes to build a peer-to-peer backup system that exploits unused disk capacity on desktop computers.

To remove the bottleneck of a single file server, it is possible to use a clustered file system in which several file servers cooperate to supply data to a single client. The Google File System [9] provides high availability and scales to hundreds of clients by providing an API that is tailored for append operations and permits direct communication between a client machine and multiple file servers. Lustre [2] uses a similar architecture in a general-purpose distributed file system. GPFS [25] is a parallel file system that makes use of multiple shared disks and distributed locking algorithms to provide high throughput and strong consistency between clients. In HYDRAstor, multiple access nodes share a common block store, but a file system currently can be modified by only a single access node.

The Frangipani distributed file system [28] has a relationship with its storage subsystem, Petal, that is similar to the relationship between HydraFS and Hydra. In both cases, the file system relies on the block store to be scalable, distributed, and highly-available. However, while HydraFS is written for a content-addressable block store, Frangipani is written for a block store that allows block modifications and does not offer duplicate elimination.

6 Future Work

While the back-end nodes in HYDRAstor operate as a cooperating group of peers, the access nodes act independently to provide file system services. If one access node fails, another access node can recover the file system and start providing access to it, but failover is neither automatic nor transparent. We are currently implementing enhancements to allow multiple access nodes to cooperate in the management of the same file system image, making failover and load-balancing an automatic feature of the front end.

Currently the file system uses a chunking algorithm similar to Rabin fingerprinting [22]. We are working on integrating other algorithms, such as bimodal chunking [13], that generate larger block sizes for comparable duplicate elimination, thereby increasing performance and reducing metadata storage overhead.

HydraFS does not yet expose snapshots to users. Although multiple versions of each file system are maintained, they are not accessible, except as part of a disaster recovery effort by system engineers. We are planning on adding a presentation interface, as well as a mechanism for allowing users to configure snapshot retention.

Although HydraFS is acceptable as a secondary storage platform for a backup appliance, the latency of file system operations makes it less suitable for primary storage. Future work will focus on adapting HydraFS for use as primary storage by using solid state disks to absorb the latency of metadata operations and improve the performance of small file access.

7 Conclusions

We presented HydraFS, a file system for a distributed content-addressable block store. The goals of HydraFS are to provide high throughput read and write access

while achieving high duplicate elimination. We presented the design and implementation of mechanisms that allow HydraFS to achieve these goals and handle the unique CAS characteristics of immutable blocks and high latency.

Through our evaluation, we demonstrated that HydraFS is efficient and supports up to 82% of the block device throughput for reads and up to 100% for writes. We also showed that HydraFS performance is acceptable for use as a backup appliance or a data repository.

A content-addressable storage system, such as HYDRAstor, provides an effective solution for supporting high-performance sequential data access and efficient storage utilization. Support for a standard file system API allows existing applications to take advantage of the efficiency, scalability, and performance of the underlying block store.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Ric Wheeler, for their comments, which helped improve the quality of this paper.

References

[1] J. Blomer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman. An XOR-based erasure-resilient coding scheme. Technical Report TR-95-048, International Computer Science Institute, Aug. 1995.

[2] Cluster File Systems, Inc. Lustre: A Scalable, High-Performance File System, Nov. 2002.

[3] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making Backup Cheap and Easy. In *Proceedings of the Fifth USENIX Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, Massachusetts, Dec. 2002.

[4] Data Domain, inc. Data Domain DDX Array Series, 2006. http://www.datadomain.com/products/arrays.html.

[5] 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 2009)*, pages 197–210, San Francisco, California, Feb. 2009.

[6] EMC Corporation. EMC Centera Content Addressed Storage System, 2006. http://www.emc.com/centera.

[7] K. Fu, M. F. Kaashoek, and D. Mazières. Fast and Secure Distributed Read-only File System. In *Proceedings of the Fourth USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, California, Oct. 2000.

[8] Filesystem in Userspace. http://fuse.sourceforge.net.

[9] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 29–43, Bolton Landing, New York, 2003.

[10] R. Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 155–162, Austin, Texas, Nov. 1987.

[11] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS Server Appliance. In *Proceedings of the Winter USENIX Technical Conference 1994*, pages 235–246, San Francisco, California, Jan. 1994.

[12] J. Katcher. Postmark: A New File System Benchmark. Technical Report 3022, Network Appliance, Inc., Oct. 1997.

[13] E. Kruus, C. Ungureanu, and C. Dubnicki. Bimodal Content Defined Chunking for Backup Streams. In *Proceedings of the Eighth USENIX Conference on File and Storage Technologies*, San Jose, California, Feb. 2010.

[14] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Systems (ASPLOS 2000)*, pages 190–202, Cambridge, Massachusetts, Nov. 2000.

[15] P. Kulkarni, F. Douglis, J. LaVoie, and J. M. Tracey. Redundancy Elimination Within Large Collections of Files. In *Proceedings of the 2004 USENIX Annual Technical Conference*, pages 59–72, Boston, Massachusetts, July 2004.

[16] U. Manber. Finding Similar Files in a Large File System. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 1–10, San Fransisco, California, Jan. 1994.

[17] R. C. Merkle. Protocols for Public-Key Cryptosystems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 122–133, Apr. 1980.

[18] A. Muthitacharoen, B. Chen, and D. Mazières. A Low-Bandwidth Network File System. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, pages 174–187, Lake Louise, Alberta, Oct. 2001.

[19] A. Muthitacharoen, R. T. Morris, T. M. Gil, and B. Chen. Ivy: A Read/Write Peer-to-peer File System. In *Proceedings of the Fifth USENIX Symposium on Operating Systems Design and Implementation (OSDI 2002)*, pages 31–44, Boston, Massachusetts, Dec. 2002.

[20] C. Policroniades and I. Pratt. Alternatives for Detecting Redundancy in Storage Systems Data. In *Proceedings of the 2004 USENIX Annual Technical Conference*, pages 73–86, Boston, Massachusetts, July 2004.

[21] S. Quinlan and S. Dorward. Venti: A New Approach to Archival Storage. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST 2002)*, Monterey, California, Jan. 2002.

[22] M. O. Rabin. Fingerprinting by Random Polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[23] 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.

[24] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carlton, and J. Ofir. Deciding When to Forget in the Elephant File System. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, pages 110–123, Charleston, South Carolina, Dec. 1999.

[25] 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 2002)*, pages 231–244, Monterey, California, Jan. 2002.

[26] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, Feb. 2003.

[27] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS File System. In *Proceedings of the USENIX 1996 Technical Conference*, pages 1–14, San Diego, California, Jan. 1996.

[28] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A Scalable Distributed File System. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 224–237, 1997.

[29] L. L. You, K. T. Pollack, and D. D. E. Long. Deep Store: An Archival Storage System Architecture. In *Proceedings of the Twenty-First International Conference on Data Engineering (ICDE 2005)*, Tokyo, Japan, Apr. 2005.

[30] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, Jan. 2004.

[31] 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*, pages 269–282, San Jose, California, Feb. 2008.

Bimodal Content Defined Chunking for Backup Streams

Erik Kruus
NEC Laboratories America
kruus@nec-labs.com

Cristian Ungureanu
NEC Laboratories America
cristian@nec-labs.com

Cezary Dubnicki
9LivesData, LLC
dubnicki@9livesdata.com

Abstract

Data deduplication has become a popular technology for reducing the amount of storage space necessary for backup and archival data. Content defined chunking (CDC) techniques are well established methods of separating a data stream into variable-size chunks such that duplicate content has a good chance of being discovered irrespective of its position in the data stream. Requirements for CDC include fast and scalable operation, as well as achieving good duplicate elimination. While the latter can be achieved by using chunks of small average size, this also increases the amount of metadata necessary to store the relatively more numerous chunks, and impacts negatively the system's performance. We propose a new approach that achieves comparable duplicate elimination while using chunks of larger average size. It involves using two chunk size targets, and mechanisms that dynamically switch between the two based on querying data already stored; we use small chunks in limited regions of transition from duplicate to non-duplicate data, and elsewhere we use large chunks. The algorithms rely on the block store's ability to quickly deliver a high-quality reply to *existence queries* for already-stored blocks. A chunking decision is made with limited lookahead and number of queries. We present results of running these algorithms on actual backup data, as well as four sets of source code archives. Our algorithms typically achieve similar duplicate elimination to standard algorithms while using chunks 2–4 times as large. Such approaches may be particularly interesting to distributed storage systems that use redundancy techniques (such as error-correcting codes) requiring multiple chunk fragments, for which metadata overheads per stored chunk are high. We find that algorithm variants with more flexibility in location and size of chunks yield better duplicate elimination, at a cost of a higher number of existence queries.

1 Introduction

Duplicate elimination (DE) is a means to save storage space. CDC techniques [25, 27, 24, 15, 3, 5] are well-established methods that use a local window (typically 12–48 bytes long) into data to reproducibly separate the data stream into variable-size chunks that have good duplicate elimination properties. Such chunking is probabilistic in the sense that one has some control over the average output chunk size given random data input. A "baseline" CDC algorithm has as primary parameters a single set of minimum, average and maximum chunk lengths, and it generates chunks of the desired size range by inspecting only the input stream. A baseline algorithm may also have less influential parameters, such as a backup cut-point policy to deal with the situations when the maximum chunk size has been reached without encountering a good cut point. In typical DE methods, one simply breaks apart an input data stream reproducibly, and then emits (stores, or transmits) only one copy of any chunks that are identical to a previously emitted chunk.

As the average chunk size of such baseline CDC schemes is reduced, the efficiency of deduplication increases. CDC schemes with average chunk sizes of around 8k have been used [25] and shown to result in reasonable deduplication. However, in storage systems, smaller chunk sizes come with costs:

- higher metadata overheads, as each chunk needs to be indexed;
- higher processing cost, which is proportional to the number of data packets processed;
- and lower compression ratio for each chunk, as compression algorithms tend to perform better on larger input.

For distributed deduplicating storage systems using error correcting codes (ECC) capable of protecting against

disk *and* node failure [12], these drawbacks are significant. Metadata needs to be associated with each ECC component of a chunk, and the indexing information used to find a block given a content hash needs to be stored redundantly; this results in higher per chunk overhead than other systems. Additionally, network costs increase as more chunks are processed. Thus, it is desirable to produce large chunks without unduly lowering the duplicate elimination ratio (DER), which we define as the ratio of the size of input data to the size of stored chunks. Note that the DER as defined takes into account both deduplication among chunks and individual chunk compression, but excludes metadata storage costs. The effect of the metadata costs can be trivially calculated; for a given metadata overhead $f \equiv {}^{\text{metadata size}}/_{\text{average chunk size}}$, the DER is reduced to ${}^{\text{DER}}/_{(1+f)}$.

In order to achieve our goal, we exploited the nature of the data stream composition produced by repeated backups. Policroniades et al. [26] noted that on real filesystems most file accesses are read-only, files tend to be either read-mostly or write-mostly, and that a small set of files generates most block overwrites. During repeated backups, entire files may be duplicated, and even when changed, the changes may be localized to a relatively small edit region. Here, a deduplication scheme must deal effectively with long repeated data segments, where our assumption for fresh data is that it have a high likelihood of reoccurring in a future backup run. The nature of the backup data led us to propose the following two principles governing possible CDC improvements for such streams:

P1. Long stretches of unseen data should be assumed to be good candidates for appearing later on (i.e. at the next backup run).

P2. Inefficiency around "change regions" straddling boundaries between duplicate and unseen data can be minimized by using shorter chunks.

In this paper, we propose algorithms that perform better than baseline algorithms under the assumption that P1 and P2 hold, and the system provides an efficient *existence query operation* that allows one to check whether a tentative chunk has been encountered in the past. By a "better" duplicate elimination algorithm, we mean one that produces a larger average chunk size than a baseline CDC algorithm while obtaining comparable DER.

P1 is justified by the fact that the amount of data modified between two backups is a small percentage of the total, and is concentrated in relatively few regions of change. P1 may in fact not be justified for systems with a high rollover of content. P1 implies that an algorithm should produce chunks of large average size when in an extended region of previously unseen data. The data is in a change region if in some vicinity of it there exist both chunks that were encountered in the past, and chunks that were not. Variations in vicinity sizes, and in how small the unseen data in a change region is chunked lead to different variants of the bimodal algorithms. Note that P2 is somewhat counter-intuitive, since it involves speculatively injecting undesirable small chunks into the storage system while providing no guarantee of an eventual storage payoff. Nevertheless, we present real-world evidence that this strategy may benefit scenarios storing many versions of an evolving data set.

Note that our bimodal chunking algorithms avoid problems with historical approaches that use resemblance detection [10, 11, 6, 4] or storage of sub-chunk information [5], whose implementations can suffer from slow speed and/or large amounts of metadata. We assume that the existence queries can be answered accurately, but discuss in Section 3.3 the effect of false positives (as could arise from the use of Bloom filters). Recently, a promising approach for efficient deduplication has been described [4] in which first a similar set of already stored chunks can be quickly selected, and then deduplication is performed within that localized environment. From the point of view of the entire system, this amounts to having a small rate of false negatives: chunks that already exist may be stored again. However, their results show that in practice the effect of these false negatives is minimal, and that they retain sufficient stream locality for good deduplication. We expect that our bimodal algorithms would also perform well in their setting, since both the fast querying algorithm and our bimodal chunking algorithms are exploiting assumptions about stream locality.

The paper is structured as follows. In Section 2 we describe baseline CDC algorithms and introduce two types of bimodal chunking improvements: splitting-apart and amalgamation algorithms. In Section 3 we begin by describing our data sets and testing tools, after which we present the results of applying the algorithms and interpret the results. We establish a performance limit for bimodal algorithms as well as briefly discussing engineering aspects. We also show that our assumptions P1 and P2 do not quite hold for our data set, yet the algorithms produced chunk sizes 2–4 times larger than those produced by a baseline algorithm with a comparable DER. Section 4 contains related work and Section 5 presents conclusions and future work.

2 Method

2.1 Using chunk existence information

Two approaches exist. In one, a breaking-apart algorithm first chunks everything with large chunks, identi-

fies change regions of new content, and then re-chunks data near boundaries of this change region at a finer level. In such an approach, a small insertion/modification of an input stream likely renders an entire large chunk non-duplicate. Were this large chunk re-chunked smaller, later occurrences of a short region of repeated change could be more efficiently bracketed.

In a slightly more flexible approach, a building-up algorithm can initially chunk at a fine level, and combine small chunks into larger ones. A building-up chunking algorithm can query for candidate big chunks at more positions, and more finely bracket such a single inserted/modified chunk. In both cases, at any point in the input stream, a decision must be made whether to emit a small chunk or a big chunk, so we refer to these algorithms as *bimodal* chunking algorithms, as opposed to the (unimodal) baseline CDC approaches.

In either approach, it is always advantageous to emit an already existing big chunk. If several big chunk emissions are possible, we emit the first-most one. Small chunks are then emitted only for non-duplicate big chunks near (adjacent to, in measurements below) duplicate big chunks. Note that in both schemes, some data may be stored in both small- and large-chunk format. In principle, this loss may be mitigated by rewriting such large chunks as two (or more) smaller chunks. However, for systems with in-line deduplication, rewriting an already emitted big chunk as two or more chunks may be impractical, so we will not consider chunk-rewriting approaches. Nevertheless, this might be possible to implement as a postprocessing step.

We target global duplicate elimination and assume that the block store can be efficiently queried for existence of chunks given a chunk content hash. Our algorithms operate in constant time per unit input, regardless of the number of stored chunks, since they require only a bounded number of chunk existence queries per chunking decision. Implementations of bimodal chunking can vary in the number and type of existence queries required before making a chunking decision. In general, we will find that the more flexibility one has in bracketing change regions and in what boundaries are allowed for large chunks, the better one's performance can be in terms of increasing chunk size.

Note that our approach does not require storing information about finer-grained blocks (e.g. non-emitted small chunks), and thus works well with any block store capable of answering whether a chunk with a given hashkey has already been stored or not. More complicated schemes, in which sub-block information is used, are possible (e.g. *fingerdiff* [5]), but the higher amount of metadata required likely leads to a higher cost of queries and makes more difficult the task of dealing with query latencies, impacting system performance

The heuristics behind our algorithms can be expected to perform well only if the backup stream has properties in line with P1 and P2. Indeed, without a similar-chunk lookup and an indirect addressing method, the first time a largely unmodified big chunk is re-chunked as small chunks, one pays the price of speculatively storing many small chunks that have no guarantee of ever being encountered again. If the small chunks re-occur sufficiently frequently in later backups (i.e. a finer grained delimiting of the duplication range), we can more than recoup the initial loss. In Section 3 we show that although P1 and P2 don't quite hold for our data set, the algorithms worked well, resulting in an average chunk size 2–4 times higher than baseline CDC for comparable DER.

2.2 Baseline rolling window cut-point selection.

Content-defined chunking works by selecting a set of locations, called *cut-points*, to break apart an input stream, where the chunking decision is based on the contents of the data itself. Typically this involves evaluating a bit scrambling function (say, a CRC) on a fixed-size sliding window into the data stream. The result of the function is compared at some number ℓ of bit locations with a predefined value, and if equivalent the last byte of the window is considered a cut-point. This generates an average chunk size of 2^ℓ, following a geometric distribution. For terseness, we will refer to such a chunker as a level-2^ℓ chunker. The probability of identifying a unique cut-point is maximized when the region searched is of size 2^ℓ.

Backup cut-points

For minimum chunk size m, the nominal average chunk size is $m + 2^\ell$. For a maximum chunk size M, a plain level-2^ℓ chunker (i.e. chunking algorithm) will hit the maximum with probability approximately $e^{-(M-m)/2^\ell}$, which can be quite frequent. Since chunking at M is no longer content-defined, the deduplication of two similar streams is commonly improved by avoiding this situation. We have adopted a simple approach of choosing a best content-defined "backup" cut-point, chunked at a level $2^{\ell-b}$, to decrease the use of these non content-defined cut-points. The data we present here has used a policy of taking the longest backup cut-point from the highest of b =2–3 backup levels; otherwise, we emit a non-content-defined chunk of maximal length. In practice, if one adopts the earliest backup cut-point, other parameters can be varied to increase the average chunk size again. This may result in a small performance improvement. More sophisticated approaches to dealing with chunks of maximum size are also possible [15].

```
1  for (each big chunk) {
2    if (isBigDup)
3      {emit as big; isPrevBigDup=true}
4    else if (isPrevBigDup || isNextBigDup)
5      {rechunk as smalls; isPrevBigDup=false}
6    else {emit as big; isPrevBigDup=true}
7  }
```

Figure 1: A simple breaking-apart algorithm.

2.3 Breaking-apart algorithms

An example of a simple breaking-apart algorithm that re-chunks a nonduplicate big chunk either before or after a duplicate big chunk is detected is shown in Figure 1.

Here the primary pass over the data is done with a large average chunk size, emitting big duplicates in line 2–3. Otherwise, in lines 4–5, a single nonduplicate data chunk after or before a duplicate big chunk is re-chunked at smaller average block size and emitted. Remaining chunks are emitted as big chunks in line 6. One can modify such an algorithm to detect more complicated definitions of duplicate/nonduplicate transitions; e.g., when N non-duplicates are adjacent to D duplicates, re-chunk R big chunks with smaller average size. Here we present results for $N = R = D = 1$, as in Fig. 1. When we varied R we found that similar results for average chunk size and DER could be obtained by simply varying the chunking parameters $\{m, 2^{\ell}, M\}$ of the baseline algorithm instead. Alternatively, one could work with the byte lengths of the chunks to limit the nonduplicate region in which small chunks are emitted adjacent to a nonduplicate/duplicate transition point.

A lookahead buffer is used to support the isNextBigDup predicate. Querying work is bounded by one query per large chunk. This is the fastest of the proposed algorithms. In Fig. 2 we illustrate the operation on a simple example input 2(a). Big chunks (b) are queried for existence (c) and we assume duplicate and non-duplicate tags are assigned as shown. All duplicate big chunks should be stored. Of the remaining chunks, the transition regions (d) are re-chunked at smaller average chunk size. The remaining non-duplicate chunks are re-emitted as big chunks (e). In the final (f) bimodal chunking, chunks 2–6 and 9–11 are of small length. Of these, note that with respect to the byte-level duplication boundaries of the input stream (a), small chunks 2, 3 and 11 are entirely within the duplicate bytes area, and may possess enhanced probabilities of recurring later. In essence, the small transition region chunks can allow the extent of duplicate bytes to be more faithfully represented.

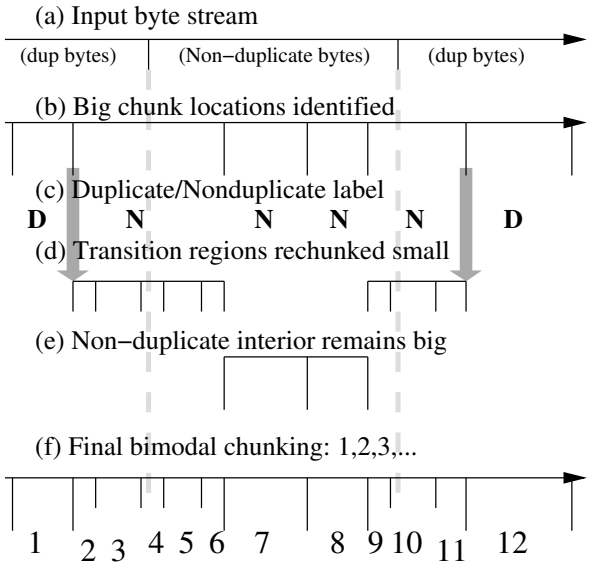


Figure 2: Breaking-apart algorithm steps.

2.4 Chunk amalgamation algorithms

Considerably more flexibility in generating variably-sized chunks is afforded by running a smaller chunker first, followed by chunk amalgamation into big chunks. Consider a simple case where big chunks are only generated by concatenation of a fixed number k of small chunks (Figure 3.) We will call these "fixed-size" big chunks because they are formed from a constant number of variably-sized small chunks during the initial forward search for big duplicates (lines 3–6). Their length in bytes is variable and their chunk endpoints are content-defined. We will call the above algorithms with fixed-size big chunks "k-fixed" algorithms. When the forward search for duplicates fails, lines 7–8 emit k chunks following a duplicate as small chunks when following a duplication region. Otherwise, those k chunks are amalgamated and emitted as a single big chunk in line 9.

A simple extension modifies lines 3–6 to allow variably-sized big chunks (1–k or 2–k small chunks) to be queried at every possible small chunk position during this decision-making process. We will label such extensions as "k-var" algorithms. With fixed-size big chunks we make at most 1 query per small chunk, while for variable-size big chunks we can make up to $k-1$ (or k) queries per small chunk.

To limit the possibility for two duplicate input streams to remain out-of-synch for extended periods, it is possible to introduce *resynchronization cut-points*: whenever the cut-point level of a small chunk exceeds some threshold (r higher than the normal chunking threshold ℓ), a big chunk can terminate there, but may never contain the resynchronization point in its interior. In this

```
1  void process( small chunks buf[0 to 2k-1] ){
2    for( pos=0; pos<=k; ++pos ) { //fwd search
3      if isBigDup(buf[pos to pos+k-1]) {
4        emit any smalls buf[0] to buf[pos-1]
5        emit big @ buf[pos to pos+k-1]
6        isPrevDupBig=true; return }
7    if( isPrevDupBig ) { emit k smalls
8      isPrevDupBig=false; return }
9    emit big @ buf[0 to k-1]; isPrevDupBig=true
10 }
```

Figure 3: A simple chunk amalgamation algorithm, in which k contiguous small chunks constitute a big chunk. Big duplicate chunks are always desirable (lines 2–6). Small chunks can only be emitted either in line 4, upon detecting an ensuing transition to duplicate data, or in line 7 when exiting a region of duplicate data. Regions considered fresh data (line 9) are emitted as big chunks.

fashion, two duplicate input streams can be forcibly resynched after a resynchronization cut-point in algorithms that do not have sufficient lookahead to do so spontaneously. This mechanism can protect against certain malicious inputs, but will lower the average chunk size. A second means to favor spontaneous resynchronization is to use a hierarchy of backup cut-points (parameter b of Section 2.2).

In our test code, we also allowed some algorithms of theoretical interest. We maintained Bloom filters for many different types of chunk emission separately: small chunks and big chunks, both emitted and non-emitted. One benefit (for example) is to allow the concept of 'duplicate' data region to include both previously emitted small chunks as well as non-emitted small chunks (that were emitted as part of some previous big chunk emission). An algorithm modified to query non-emitted small chunks (i.e. the small chunks that were not emitted because they were part of some big chunk) can detect duplicate data at a more fine-grained level, at the cost of additional storage for such sub-chunk metadata. Nevertheless, when resources are more plentiful, implementations such as *fingerdiff* adopt such an approach and obtain substantial compression improvements [5].

Figure 3 shows the algorithm as applied in this paper. The length of the lookahead buffer is of minimal size and gives the behavior that transition regions are never covered by more than k small chunks. It is also quite reasonable to extend the lookahead to $3k-1$ chunks, and allow up to $2k-1$ small chunks to precede an upcoming duplicate big chunk, as depicted in Fig. 4

The logic of breaking apart and amalgamation algorithms (Figs. 2 and 4) is highly similar. For amalgamation input 4(a), small chunks (b) are used to form big chunks that are defined here as exactly 3 consecutive

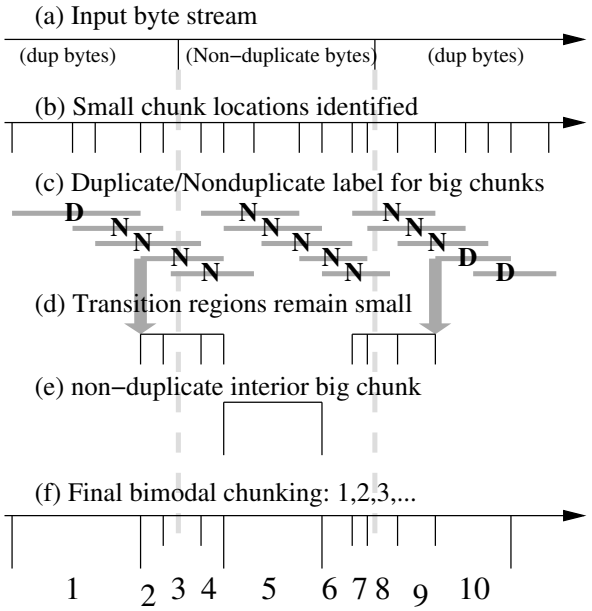


Figure 4: "k-fixed" amalgamation algorithm steps. We assume fixed-size big chunks are constituted of precisely three small chunks in this example.

small chunks. Big chunks are queried in 2/4(c) and first-most-occurring duplicate big chunks are emitted. Of the remaining chunks, transition regions 2/4(d) are emitted as small chunks. The remaining non-duplicate interior chunks are re-emitted as a series of big chunks inasmuch as possible 2/4(e), with one straggling small chunk left over at the end in 4(e). The final chunk emission 4(f) has small chunks 2–4 and 6–9. With the byte-level duplication points as in 4(a), small chunks 2 and 9 lie entirely within the span of duplicate bytes, and may have enhanced potential for deduplication.

Querying work is larger for amalgamation algorithms than for breaking-apart. Breaking apart uses one query per big chunk, whereas k-fixed amalgamation uses up to k queries per big chunk (one per small), and k-var amalgamation for big chunks consisting of 2–k small chunks uses up to $k(k-1)$ queries per big chunk. The increased number of existence queries for k-var amalgamation may be unattractive for practical implementations.

3 Results and Discussion

3.1 Test data

We used a data set for testing consisting of 1.16 Terabyte of full Netware backups of hundreds of user directories over a 4 month period. For privacy reasons, we had no idea what the distribution of file types was, only that it was a large set of real data, typical of what might be seen

in practice. Some experiments were also conducted using an additional 400 GB of incremental backups during this same period, but the results reported here include only the data from the full backups.

In order to study the behavior of the algorithms on data sets with characteristics different from our 1.16 TB data, we also analyzed data sets similar to those of Bobbarjung et al. [5], consisting of tar files for consecutive releases of several large projects. Their work targeted improvements for very small chunk sizes (< 1KB), while we target large chunk sizes.

3.2 Simulation tools

We have developed a number of tools for offline, anonymized, analysis of very large customer data sets. The key idea was to generate a binary "summary" of the input data, storing fine-grained information about potential chunk-points that could later be reused to generate any coarser-grained re-chunking. For every small chunk generated with expected size 512 bytes, we stored the SHA-1 hash of the chunk, as well as the chunk size and actual cut-point level ℓ (# of terminal zeroes in the rolling window hash). The summary data was obtained by running with minimum chunk size 1 byte and maximum chunk size 100k, with expected chunk size 512 bytes. This chunk data was sufficient to re-chunk our input data sets. Data sets that generate no chunk-points at all (e.g. all-zero inputs) are better handled by reducing the maximum chunk size used for generating the summary stream.

Our utilities also stored local compression estimates, generated by running every fixed-size chunks (ex. 4k, 8k, 16k, 32k) through LZO and storing a single byte with the percent of original chunk size. Then, given the current file offset and chunk size, we could estimate the compression at arbitrary points in the stream. Using piecewise constant or linear approximations for the estimated size of compressed chunks yielded under 1% errors in compressed DER for our large dataset. In this fashion, the 1.16 Terabyte input data could be analyzed as a more portable 60 GB set of summary information (a sequence of several billion summary chunks, involving over 400 million distinct chunks). Such re-analyses took hours instead of days. We also stored, to a separate file, the duplicate/nonduplicate status of every summary stream chunk as it was encountered. This allowed us to investigate the size distribution of nonduplicate and duplicate segments of input data, as well as efficiently ascertaining which small-chunk decisions would *later* generate duplicate chunks.

To answer existence queries we used in-memory Bloom filters of up to 2 Gigabytes in length. The summary streams and Bloom filters allowed us to quickly simulate a large number of chunking algorithms on up to 1.5 Terabytes of original raw data using a single computer. We were also interested in knowing the limits of coalescing small chunks into large chunks. Since an exact calculation is prohibitive, a simple approximation was obtained by coalescing all always-together chunk sequences into single chunks. Other tools allowed us to consult an oracle in order to maintain statistics about the future re-encounter probabilities of different types of chunks.

Because of intended use at customer sites, the tools were also used to evaluate faster alternatives to Rabin Fingerprinting [7, 29] to select cut-points. Using a combination of boxcar functions and CRC-32c hashes allowing input streams to be chunked at memory bandwidth and represented a considerable time savings when generating chunking summaries. We verified that using a faster rolling window (operating essentially at memory bandwidth) had no effect upon DER, corroborating Thaker's [31] observation that with typical data even a plain boxcar sum generated a reasonably random-like chunk size distribution. He explained this as a reflection of there being enough bit-level randomness in the input data itself, making a high-quality randomizing hash function unnecessary in practice. We verified that choice of rolling window function had no little impact upon DER measurements for our 1.16 TB dataset.

3.3 DER of different chunking algorithms

Within a given algorithm, there are several parameters, such as minimum m and maximum M chunk size, and trigger level ℓ, which can generate different behavior. Breaking-apart and amalgamation algorithms also have other parameters, such as k (the number of small chunks in a big chunk) and an optional resynchronization parameter r (defining a coarser-grained chunking level $2^{\ell+r}$ across which no big chunk may extend). When an algorithm was run over the entire 1.16 Terabyte data set or its summary, we measured the DER as the ratio of input bytes to bytes within stored chunks. Bytes within stored chunks could be reported raw, or as compressed size estimates. We used an LZO compressor to derive compression values; however, other compressors should display qualitatively similar behavior. Compression is relevant because most archival systems store data in compressed format. We explored a wide space of parameters for amalgamation (fixed- and variable-size big chunks) and breaking-apart algorithms on this data set. We show plots assuming zero metadata overhead initially and will give an illustration of the effects of metadata upon the DER later.

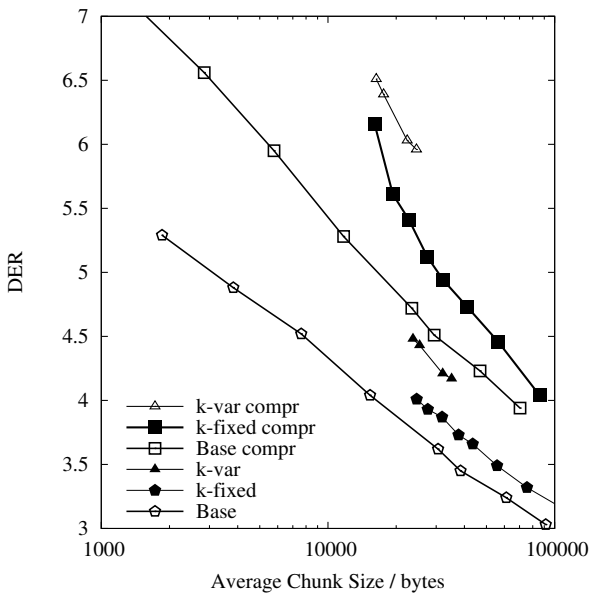


Figure 5: Performance of two amalgamation chunking algorithms, *k*-fixed and *k*-var, compared to a baseline chunking algorithm "Base", over a range of chunk sizes. The top 3 "compr" curves are the same data as the lower three traces, but DER and chunk sizes are reported assuming compressed chunk storage.

Performance of bimodal amalgamation chunking

Figure 5 compares two bimodal amalgamation algorithms "*k*-fixed" and "*k*-var" with standard baseline chunking algorithms "Base". For each of these 3 chunking algorithms, raw DER values and chunk sizes are in the bottom 3 traces, while corresponding DER using stored compressed chunk sizes appears in the upper 3 traces. Comparing the two sets of three traces, we note for compressed storage the traces are more highly sloped, which reflects the rapid initial rise in compression efficiency as chunk size is increased. Linearity in the raw DER traces indicate some scale-independent statistical behavior in our large archive dataset: this is not the case for some small test datasets that we present later.

In this and later figures, precise parameter settings of a particular algorithm are usually not influential, serving to move measured points along the same general curve. Since precise parameter settings are not crucial, the parameters we do describe should be viewed as examples of reasonable settings.

The "Base" baseline chunking traces shown in Fig. 5 varied the minimum, nominal average and maximum chunk sizes $\{m, m+2^{\ell}, M\}$, often maintaining a 1:2:3 ratio for these values. We consulted $b = 3$ levels of backup cut-points if maximum chunk size was encountered.

The "*k*-fixed" traces of Fig. 5 use an amalgamation algorithm, running with fixed-size big chunks (i.e. a big chunk consists always of k small chunks). Half these runs maintained a 1:2:3 ratio for min:avg:max, with $k = 8$ and $r = 4$. Two used $k = 4$ instead, and two did not use resynchronization points. Investigating more parameter settings showed that minor variations in chunking parameters typically lay along the same curve: the algorithm was robust to parameter choices. We found a broad optimal region for k from 8 to 12, and suggest that resynchronization points be either unused or maintained at $r \gtrsim 3$.

The algorithm labelled "*k*-var" in Fig. 5, at an additional querying cost, allows variable-sized big chunks that use any number 1–k of small chunks. It also used Bloom Filter queries for small chunks which were previously encountered but emitted only as part of a previous big chunk as finer-grained delineators of change regions. In spirit the "*k*-var" traces of Fig. 5 might be viewed as a lower bound for what more sophisticated algorithms using sub-chunk information (such as *fingerdiff* [5]) or chunk rewriting approaches could achieve.

Later, we will show that the extensions to the "*k*-var" algorithms provide only slightly better performance. This suggests that the most important algorithmic difference between fixed- and variably-sized big chunks lay in the increased flexibility of generating and recognizing large chunks. Nevertheless, algorithms in this "*k*-var" class require more existence queries so they are not algorithms of choice.

Note that the "*k*-fixed" algorithm of Fig. 5 can already maintain average compressed chunk sizes up to 3–4× as large as a baseline chunker at small chunk sizes (e.g. DER 6.1 at 16100 bytes using $k = 4$ and no resynchronization, as compared to an interpolated 4700 bytes for "Base compr"). For uncompressed storage systems, we see that *k*-fixed bimodal amalgamation algorithms uniformly yielded ≈50% increase in average uncompressed chunk size, even at the largest (96k) chunk sizes presented.

Our implementation used a look-ahead buffer of $2k$ small chunks and in-memory Bloom filters for speed. As noted before, a lookahead buffer of $3k-1$ chunks is also a reasonable choice. In practice, however, to maintain streaming performance very much larger look-ahead buffers may be necessary, since answering existence queries is likely to require asynchronous network or disk operations of high latency.

Our use of Bloom filters in answering existence queries led us to question the impact of false positives. For the "*k*-fixed" amalgamation algorithm, we found all benefits of bimodal chunking over the baseline were negated by ≈2.5% false positives. Falsely identified duplicate/nonduplicate transitions should be avoided. So techniques such as a hierarchy of more accurate Bloom filters [39] may be useful. Alternatively, in other work,

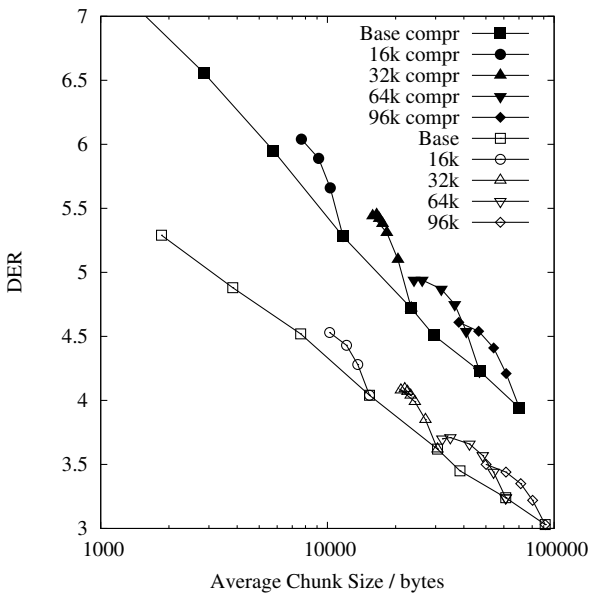


Figure 6: Breaking-apart chunking algorithms compared with baseline performance.

we have adapted efficient hash table implementations [19, 16, 23] to take full advantage of SSD R/W characteristics (possibly in conjunction with fingerprint approaches) to provide fast, exact answers to existence queries.

Variants of amalgamation algorithms, that prioritize equivalent choices of big chunk if they occurred, were found to offer no significant performance improvement. In fact, several such attempts work badly when run on actual data, often for rather subtle reasons.

Small chunk statistics, using an oracle

Using knowledge of the full set of small chunk emissions we investigated the statistics of the smaller transition region chunks, which bore out premise P2 for an amalgamation algorithm using fixed-size big chunks. For example (not shown in figures), for $k = 8$ small chunks in a transition region between two duplicate big chunks, the bordering small chunks have around 88% chance of being encountered subsequently, dipping to 86% for central small chunks. For one-sided duplication transitions, we found that the small-chunk duplication chance decayed from ~75% to ~67%. Bimodal chunking with $k = 32$ showed small-chunk duplication probability declining from 86% adjacent to the duplicate big chunk to 65% at the furthest small chunk. These experimental results agree with earlier expectations based on Fig. 4 assuming good future duplication of byte-level duplication regions and, say, a uniform location for the start of the byte-level non-duplicate region in 4(a) with respect to the small chunk transition region 4(d).

Performance of bimodal breaking-apart chunking

In Figure 6 we present results with a breaking-apart algorithm, which uses one query per large chunk, compared to the baseline algorithm. Most runs retain baseline $m : m + 2^{\ell} : M$ settings in a 1:2:3 ratio. Beginning with a baseline chunker we consecutively divided these settings by two to generate a series of small chunkers, which were used in the breaking apart algorithm of Fig. 1. A few additional points vary R, the size of transition region that gets re-chunked, but do not depart substantially from the breaking-apart curves for $R = 1$. We note that reasonable performance is obtainable by choosing a small chunker with average chunk size about 4–8 times smaller than the original baseline chunker.

Comparing Figs. 5 and 6, we see that a carefully tuned breaking apart algorithm can be competitive with the performance of amalgamation algorithms with fixed-size big chunks, particularly in the regime of chunk sizes $\gtrsim$40k. The practical benefit of breaking-apart over the "k-fixed" amalgamations of Fig. 5 is a reduction in the number of existence queries by a factor of k.

Effect of non-zero metadata overhead

One approach to accounting for metadata effects is to pretend that it simply increases the average stored block size by some number of bytes. Another instructive approach is to consider the the metadata effects on the oft-reported DER values. For example, with a metadata overhead of 800 bytes per chunk, we can use the known total amount of input bytes (which is a constant 1.16 TB in Figs. 5 and 6) to transform the DER value of each measurement, while still reporting the average size of the chunk.

In Figure 7, we have simply scaled the DER values of the empty symbols, which are traces taken from Fig. 5, by reducing their DER by $1 + f$. Here $f \equiv {}^{\text{metadata size}}/_{\text{average chunk size}}$ is the metadata overhead, and the transformed traces are plotted with solid symbols. The DER reduction can be quite dramatic at low chunk sizes where metadata overhead is a substantial fraction of the stored chunk size. We see that including metadata magnifies the DER improvement relative to a baseline chunker of equivalent average chunk size. The figure motivates maintaining average chunk sizes much larger (preferably $\gtrsim 20\times$) than the per-chunk metadata overhead.

Data	# of versions	Baseline chunk size / bytes	Baseline DER	Amalgamation chunk size / bytes	Amalgamation DER	Compressed size of 16k records / 16k
gcc source	20	4952	4.68	13742	4.59	0.37
gdb source	10	6184	4.14	15225	4.05	0.35
linux source	10	6921	3.51	16804	3.52	0.40
emacs source	10	7525	3.23	17265	2.95	0.46

Table 1: Comparison of DER (w/ LZO) achieved by baseline chunkers and amalgamation algorithms. The average input chunk size of the baseline chunker was 16k with allowed sizes 8k–24k and two backup levels. The amalgamation used large chunks composed of exactly $k = 8$ small chunks. Values of chunk size and DER reflect chunks stored in compressed LZO format. The average compressibility of fixed-length 16k records of input data (no deduplication) are in the last column.

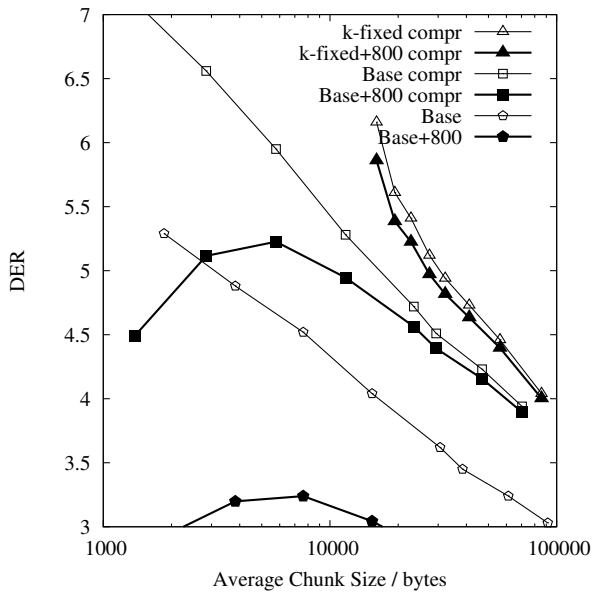


Figure 7: Two baseline and one "*k*-fixed" amalgamation algorithm curves (open symbols) from Fig. 5 have been transformed (solid symbols) to reflect 800 metadata bytes per chunk.

Performance using source code archives

We also analyzed data sets consisting of tar files for consecutive releases of several large projects. The compressed chunk size and DER under one set of baseline conditions and an amalgamation algorithm based upon these small chunks is shown in Table 1. We see that amalgamation has increased the average chunk size of stored chunks by a factor of around 2.5, with a worst case decrease in DER of 8%.

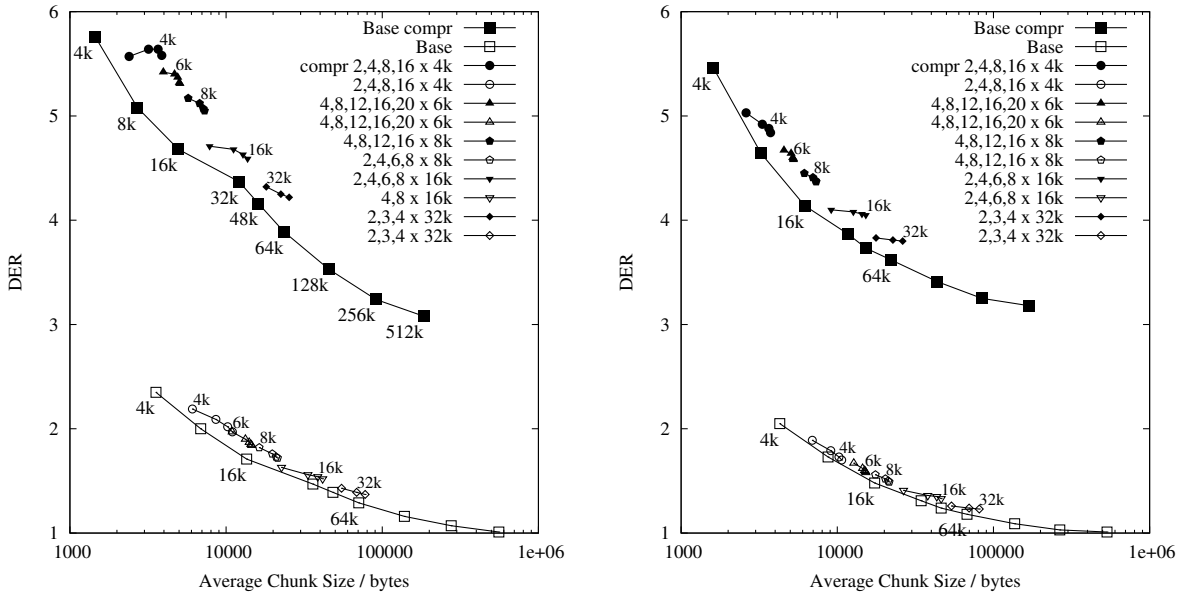
A picture of the performance of baseline and "*k*-fixed" amalgamation on these source archives is offered by Fig. 8, which shows DER curves with compression (top curves) and without (bottom). Corresponding to various baseline chunkers, we ran "*k*-fixed" amalgamate algorithms as in Fig. 5 for k values between 2 and 20. Recall that $k = 8$ was suggested to be a reasonable value for the large dataset. Improvements in DER and chunk size are much worse for these small archive datasets, when compared with the 1.16 TB dataset of Fig. 5.

The baseline chunkers all display uncompressed DER that approaches 1.0 as average chunk size rises, showing that at large chunk sizes, DER can be obtained primarily by using compression. These data sets have small file sizes and quite scattered change sections (i.e. property P1 for filesystems may not apply well when the density of changes is large and somewhat uniform). The DER (w/o LZO) points are usually above (better) the smooth Baseline curve, but do not show significant improvement. The improvement is better when storage of compressed chunks is considered. The emacs data set consistently shows the smallest improvements from amalgamation, as well as the least duplicate elimination (2.0 at 4k average chunk size, 4.12 compressed) and least compressibility (fixed-size 16k chunks were compressed to 46% of their original length).

Even though there is no reason that tar files of source code releases should concentrate most change regions into a small subset of files, amalgamation still shows modest DER vs. chunk size improvement with respect to baseline CDC chunking. Lightly degraded DER was achieved with average chunk sizes larger by factors of 2.5× (see Table 1) in these data sets, as compared to a factor of 3–4× in the actual 1.16 TB archival data set.

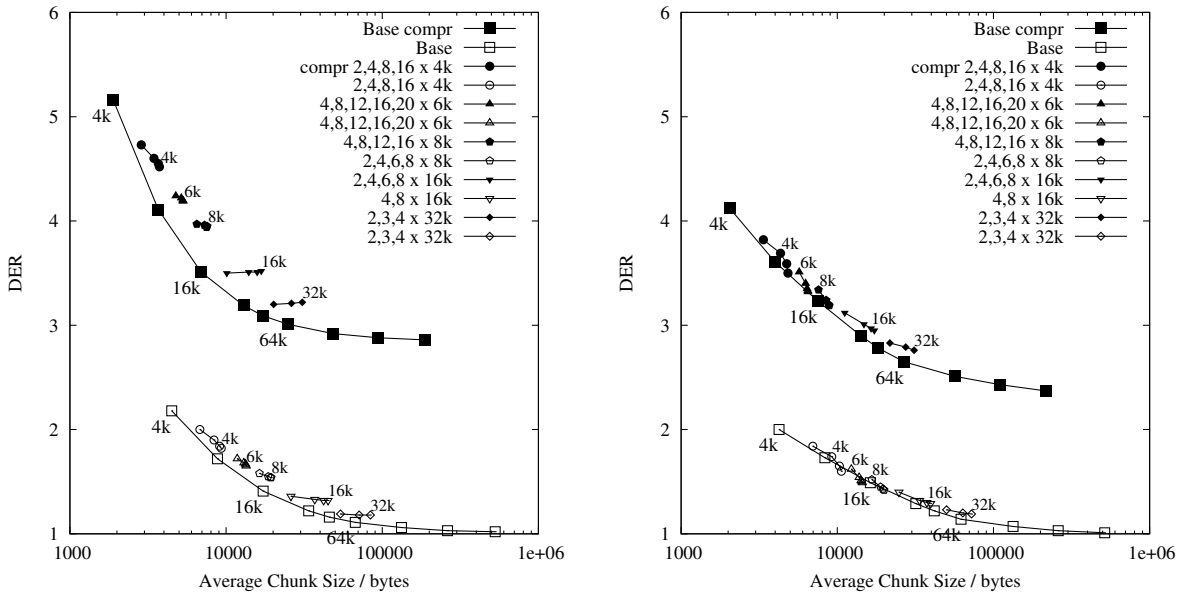
Optimal "always-together" chunks

For our 1.16 TB data set, it is also interesting to consider what a good theoretical amalgamation of small chunks would be. A simple set of optimization moves is to always amalgamate consecutive chunks that always occurred together. This will not affect the DER at all, but will increase the average chunk size. Iterating this pro-



(a) DER vs. chunk size: gcc dataset

(b) DER vs. chunk size: gdb dataset



(c) DER vs. chunk size: linux dataset

(d) DER vs. chunk size: emacs dataset

Figure 8: Duplicate elimination versus stored chunk size measurements on consecutive source code releases. Baseline and bimodal k-fixed chunking were performed, yielding results for uncompressed storage (lower traces, open symbols) and compressed storage (upper traces, solid symbols). Chunk compression used the default LZO settings. Bimodal series denoted in the legends as "$k_1, k_2, \ldots$ x Nk" amalgamate a fixed number, k, of chunks output from the baseline chunker with Nk average chunk length.

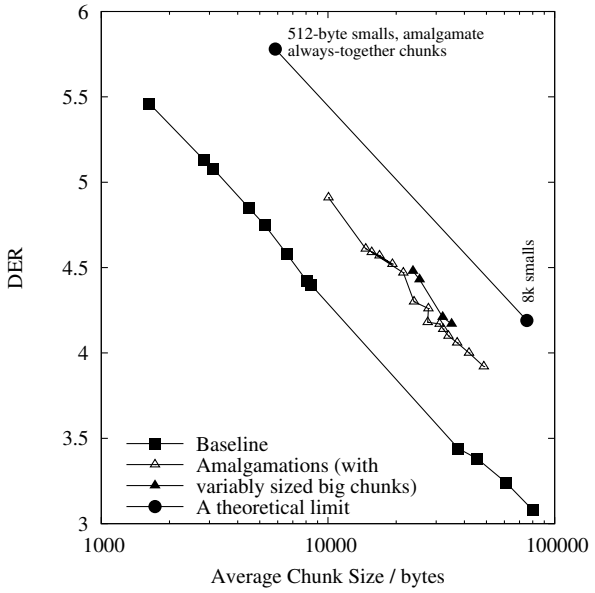


Figure 9: Baseline and k-var amalgamation are compared with theoretical chunk size limits determined by amalgamating every set of chunks which always co-occurred in our 1.16 Terabyte data set. k-var amalgamation results (triangles) cover a wide range of parameters chunking parameters. Solid triangles in Figs. 5 and 9, using extensions to the basic algorithm, are included here for comparison.

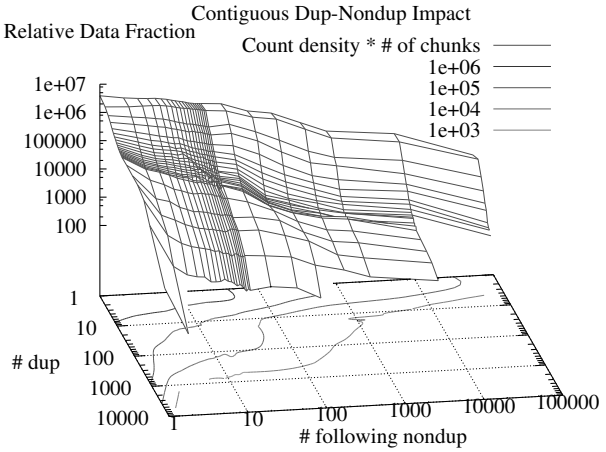


Figure 10: Histogram of number of contiguous duplicate chunks vs. number of subsequent contiguous nonduplicate chunks at the 512-byte expected chunk size. Raw counts have been scaled by the number of chunks to produce histogram values representing the total amount of input data. Note the logarithmic scales: the overwhelmingly most frequent (and still most important with regard to total amount of input data involved) occurrence is one duplicate chunk followed by one nonduplicate chunk.

duces that longest possible strings of chunks that always co-occurred and increases the average chunk size. This parallelized calculation is lengthy and non-scalable.

Using "future knowledge" to amalgamate all always-together chunks was done for input chunk sequences of 512 and 8192 average size to produce two isolated points in Fig. 9. Analyzing the raw summary stream, with chunks 512 bytes long on average, increased the average uncompressed stored chunk size from 576 to 5855 bytes (i.e. the average number of always-co-occurring small chunks was around 10 for this data set). Similarly, the other theoretical calculation increase the average chunk size from around 8k to 75k bytes, once again nearly a factor of 10× improvement in uncompressed chunk size.

In practice, amalgamating often- or always-together chunks opportunistically may be a useful background task to optimizing storage. This experiment provides an easily-defined theoretical bound against which we can judge how well our simple algorithms based on duplicate/nonduplicate transition regions were performing: 10× improvement can be achieved, with such an oracle.

For comparison, Fig. 9 also presents a number of amalgamation results with variable-size big chunks (k-1 queries per small chunk). Such amalgamation algorithms come almost half-way from the baseline curve to this particular theoretical limit. These runs had a haphazard selection of m, ℓ and M small chunk size settings, use 0–4 resynchronization cut-points (usually zero or 4), and mostly have $k = 8$. Again, noting that the results lie more or less along a common line we conclude that precise values of parameter settings are not vitally important. We also note that performance is on par with the traces labeled "k-var" in Fig. 5 (reproduced here in Fig. 9 as solid triangles). This indicates that the additional complication of using sub-chunk information to delineate change regions was not particularly useful.

3.4 Data characteristics

Size-of-modification distribution

Although originally formulated based on considerations of simple principles P1 and P2, it is important to judge how much our real data departs from such a simplistic data model. We found that the actual data deviated quite substantially from an "ideal" data set adhering to P1 and P2. A simplest-possible data set adhering to P1 might be expected to have long sequences of contiguous nonduplicate data during a first backup session, followed by long stretches of duplicate data during subsequent runs.

We interrogated the anonymized summary stream, as chunked at the 512-byte expected chunk size, using a bit-

stream summary of the "current" duplication status of the chunk. The actual histograms of number of contiguous nonduplicate chunks vs. number of contiguous duplicate following chunks (and vice-versa) showed an overwhelming and smoothly varying preference to having a single nonduplicate chunk followed by a single duplicate chunk. A 2-dimensional histogram of the final contiguous numbers of duplicate/nonduplicate chunks (after 14 full backup sessions) is in Figure 10. The histograms after the first "full" backup was of similar character. Such histograms do not suffice for estimating DER since duplication counts are absent. This analysis found no naive adherence to P1 and P2.

Only a minor fraction of the input stream was data occurring as long stretches of unseen data. Only the earlier oracular results provided direct evidence for P2: small chunks close to duplicate big chunks did indeed have significantly augmented re-emission probabilities. This effect can be predicted simply by assuming a uniform location of the transition region from duplicate to nonduplicate bytes within the large chunk being stored as smaller chunks in Figs. 2(d) and 4(d), and may be the dominant reason why bimodal chunking works for archival data.

This suggests that for input data sets showing such high interspersal of duplicate with nonduplicate chunks, alternate approaches may be able to come closer to the theoretical limit than the algorithms presented in this paper. Nevertheless, even for such data, even simple bimodal chunking heuristics were able to increase average chunk size by a factor of 3 or more.

4 Related Work

For our purposes, the speed of blocking (chunking) was a consideration because we target throughputs of several hundred MB/s. The simplest and fastest approach is to break apart the input stream into fixed-size chunks. This is the approach taken in the *rsync* file synchronization tool [34, 33]. However, consider what happens when an insertion or deletion edit is made near that beginning of a file: after a single chunk is changed, the entire subsequent chunking will be changed. A new version of a file will likely have very few duplicate chunks. Pratt [26] provides good comparison of fixed- and variable-sized chunking for real data. Lufei et al. [22] provides an introduction to options such as gzip, delta-encoding, fixed-size blocking and variable-size chunking. For filesystems, You et al. [36] compares chunking and delta-encoding. Delta-encoding is particularly good for things like log files and email, which are characterized by frequent small changes.

CDC produces chunks of variable size that are better able to restrain changes from a localized edit to a limited number of chunks. Applications of CDC include network filesystems of several types [2, 27], space-optimized archival of collections of reference files [9, 14, 37], as well as file synchronization [32, 15]. By using special rolling window functions in innermost loops, the baseline CDC algorithms can operate very quickly.

Mazières' Low-Bandwidth File System (LBFS) [25, 31] was influential in establishing CDC as a widely used technique. Usually, the basic chunking algorithm is typically only augmented with limits on the minimum and maximum chunk size. More complex decisions can be made if one reaches the maximum chunk size [30, 13, 15] (see Section 2.2).

Alternatives to CDC for compressing data exist and typically have higher cost. An often used technique in more aggressive compression schemes is resemblance detection and some form of delta encoding. Unfortunately, finding maximally-long duplicates [17, 18, 1] or finding similar (or identical) files in small [5] or large (gigabyte) [8, 10, 20, 11, 28] collections is a nontrivial task.

In HYDRAstor [12] and DEBAR [35], existence queries (and *global* deduplication) can be addressed efficiently by consulting a scalable, distributed data structure. Our approach has been to tackle the small chunk size problem directly. A noted in the introduction, a recent alternative approach is to reduce metadata requirements by practicing only *local* duplicate elimination within a suitably large local basin of data. For example, the approach of Brin et al. [6] has been revived in an elegant "extreme binning" approach that distributes information at a large-block level (file-level representative hash) to detect near-similarity, and has been shown to achieve near-optimal deduplication at small-chunk level [4]. Another recent approach describes a sparse indexing approach to determining similar segments of an stream [21].

Bimodal chunking presumes only an existence query for already-stored chunks, and has the potential to provide system improvements of several types. The increase in average chunk size (roughly 2.5× in these data sets, and 3–4× in the 1.16 TB archival data set) decreases the storage cost for metadata describing these chunks. By reducing the number of disk accesses, there are potential increases in read and write speeds as fewer transactions with the storage units are involved. Furthermore, the existence query information can be used in some backup systems to entirely elide network transmission of existing duplicates, which may result in additional write speed improvements or decreased system cost.

5 Conclusion and Future Work

In this paper, we proposed bimodal algorithms that vary the expected chunk-size dynamically. They are able to

perform content-defined chunking in a scalable manner, involving a constant number of chunk existence queries per unit of input. Significantly, these algorithms require no special-purpose metadata to be stored. We show that these algorithms increased average chunk size while maintaining a reasonable duplication elimination ratio. We demonstrated the benefits of the algorithms when applied to 1.16 Terabyte of actual backup data as well as to four sets of source code archives.

Although the statistics of these data sets suggest that they do not conform to our expectations based on principles P1 and P2, the algorithms still perform well, leading us to conjecture that they are robust (applicable to many types of archival inputs). We expect the proposed algorithms will behave best for storage of versioned data in block stores with high metadata cost, but we plan to evaluate them for other data sets.

Under a wide variety of chunking parameters, chunk amalgamation algorithms performed well. They present more flexibility in querying for duplicate chunks than algorithms involving breaking apart chunks within a preliminary large chunking. We also plan to investigate algorithms that use compressibility to govern chunking decisions based on fast entropy estimation.

This work has targeted evaluating a prospective bimodal chunking algorithm that has potential to address real issues in the HYDRAstor storage system and other systems that require large per-chunk storage overhead. The simple algorithms of Figs. 1 and 3 used in the evaluation are in the process of being adapted for inclusion and evaluation in HYDRAstor. Because of the latency of answering existence queries, this requires a larger lookahead buffer and issuing (in a straightforward approach) all possible existence queries. Additionally, current storage systems go to great lengths to avoid disk accesses . For example, both HYDRAstor and Data Domain products address disk access reduction and locality of access issues and both have used Bloom filters to reduce disk the number of disk accesses [38]. Because of the disk bottleneck, efficient mechanisms to reply to existence queries with minimal impact of streaming read and write performance is desired. Implementation, currently underway for the HYDRAstor storage product, may eventually involve new data structures, or even new hardware (particularly SSDs) before bimodal chunking becomes a commercial offering.

6 Acknowledgments

We would like to thank our shepherd, Randal Burns, whose feedback has greatly improved the paper, and the anonymous reviewers for their comments and suggestions. We also wish to acknowledge Krzysztof Lichota for his work developing fast rolling windows, using boxcar functions, to obtain throughputs higher than those achievable with the usual approach of Rabin Fingerprinting [7, 29] to select cut-points.

References

[1] AGARWAL, R. C. Method and computer program product for finding the longest common subsequences between files with applications to differential compression. United States Patent 20060112264, May 2006.

[2] ANNAPUREDDY, S., FREEDMAN, M., AND MAZIÈRES, D. Shark: Scaling File Servers via Cooperative Caching. In *NSDI '05 Paper [NSDI '05 Technical Program]* (2005).

[3] BARRETO, J., AND FERREIRA, P. A replicated file system for resource constrained mobile devices. In *Proceedings of IADIS International Conference on Applied Computing* (2004).

[4] 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 2009)* (Sept. 2009).

[5] BOBBARJUNG, D. R., JAGANNATHAN, S., AND DUBNICKI, C. Improving duplicate elimination in storage systems. *Trans. Storage 2*, 4 (2006), 424–448.

[6] BRIN, S., DAVIS, J., AND GARCIA-MOLINA, H. Copy detection mechanisms for digital documents. In *In Proceedings of the ACM SIGMOD Annual Conference* (1995), pp. 398–409.

[7] BRODER, A. Some applications of Rabin's fingerprinting method. *Sequences II: Methods in Communications, Security, and Computer Science* (1993), 143–152.

[8] CHOWDHURY, A., FRIEDER, O., GROSSMAN, D., AND MCCABE, M. C. Collection statistics for fast duplicate document detection. *ACM Trans. Inf. Syst. 20*, 2 (2002), 171–191.

[9] DENEHY, T., AND HSU, W. Duplicate management for reference data. Technical report RJ 10305, IBM Research, October 2003.

[10] DOUGLIS, F., AND IYENGAR, A. Application-specific Delta-encoding via Resemblance Detection. In *Proceedings of the USENIX Annual Technical Conference* (2003).

[11] DOUGLIS, F., KULKARNI, P., LAVOIE, J. D., AND TRACEY, J. M. Method and apparatus for data redundancy elimination at the block level. United States Patent 20050131939, June 2005.

[12] 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* (2009), USENIX Association, pp. 197–210.

[13] ESHGHI, K., AND TANG, H. K. A framework for analyzing and improving content-based chunking algorithms. Technical report HPL-2005-30R1, HP Laboratories, 10 2005.

[14] FORMAN, G., ESHGHI, K., AND CHIOCCHETTI, S. Finding similar files in large document repositories. In *KDD '05: Proceeding of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining* (New York, NY, USA, 2005), pp. 394–400.

[15] GUREVICH, Y., BJORNER, N. S., AND TEODOSIU, D. Efficient chunking algorithm. United States Patent 20060047855, March 2006.

[16] HUA, N., ZHAO, H., LIN, B., AND XU, J. Rank-indexed hashing: A compact construction of bloom filters and variants. In *IEEE International Conference on Network Protocols (ICNP 2008)* (Oct. 2008), pp. 73–82.

[17] JAIN, N., DAHLIN, M., AND TEWARI, R. TAPER: Tiered Approach for Eliminating Redundancy in Replica Sychronization. In *USENIX Conference on File and Storage Technologies (FAST05)* (Dec 2005).

[18] JAIN, N., DAHLIN, M., AND TEWARI, R. TAPER: Tiered Approach for Eliminating Redundancy in Replica Synchronization. Tech. rep., Technical Report TR-05-42, Dept. of Comp. Sc., Univ. of Texas at Austin, 2005.

[19] KANIZO, Y., HAY, D., AND KESLASSY, I. Optimal fast hashing. In *28th IEEE International Conference on Computer Communications (INFOCOM)* (Apr. 2009), pp. 2500–2508.

[20] KULKARNI, P., DOUGLIS, F., LAVOIE, J., AND TRACEY, J. Redundancy Elimination Within Large Collections of Files. In *Proceedings of the USENIX Annual Technical Conference* (2004).

[21] LILLIBRIDGE, M., ESHGHI, K., BHAGWAT, D., DEOLALIKAR, V., TREZISE, G., AND CAMBLE, P. Sparse indexing: large scale, inline deduplication using sampling and locality. In *Proceedings of the 7th conference on File and storage technologies* (2009), USENIX Association, pp. 111–123.

[22] LUFEI, H., SHI, W., AND ZAMORANO, L. On the effects of bandwidth reduction techniques in distributed applications. *Proceedings of International Conference on Embedded and Ubiquitous Computing (EUC'04)* (2004).

[23] LUMETTA, S., AND MITZENMACHER, M. Using the power of two choices to improve bloom filters. *Internet Mathematics 4*, 1 (2007), 17–34.

[24] MOULTON, G. H. System and method for unorchestrated determination of data sequences using sticky byte factoring to determine breakpoints in digital sequences. United States Patent 6810398, October 2004.

[25] MUTHITACHAROEN, A., CHEN, B., AND MAZIÈRES, D. A low-bandwidth network file system. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2001), pp. 174–187.

[26] POLICRONIADES, C., AND PRATT, I. Alternatives for detecting redundancy in storage systems data. In *USENIX 04: Proceedings of the USENIX Annual Technical Conference* (2004).

[27] PORTS, D. R. K., CLEMENTS, A. T., AND DEMAINE, E. D. PersiFS: a versioned file system with an efficient representation. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles* (New York, NY, USA, 2005), pp. 1–2.

[28] PUGH, W., AND HENZINGER, M. H. Detecting duplicate and near-duplicate files. United States Patent 6658423, December 2003.

[29] RABIN, M. Fingerprinting by random polynomials. Technical report TR-15-81, Harvard University, 2003.

[30] SCHLEIMER, S., WILKERSON, D. S., AND AIKEN, A. Winnowing: local algorithms for document fingerprinting. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2003), pp. 76–85.

[31] SPIRIDONOV, A., THAKER, S., AND PATWARDHAN, S. Sharing and bandwidth consumption in the low bandwidth file system. Tech. rep., Department of Computer Science, University of Texas at Austin, 2005.

[32] SUEL, T., NOEL, P., AND TRENDAFILOV, D. Improved File Synchronization Techniques for Maintaining Large Replicated Collections over Slow Networks. In *ICDE '04: Proceedings of the 20th International Conference on Data Engineering* (Washington, DC, USA, 2004), p. 153.

[33] TRIDGELL, A. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, April 2000.

[34] TRIDGELL, A., AND MACKERRAS, P. The rsync algorithm. Technical report TR-CS-96-05, Australian National University, Deparment of Computer Science, FEIT, ANU, 1996.

[35] YANG, T., JIANG, H., FENG, D., AND NIU, Z. DEBAR: A Scalable High-Performance De-duplication Storage System for Backup and Archiving. *CSE Technical reports* (2009), 58.

[36] YOU, L., AND KARAMANOLIS, C. Evaluation of efficient archival storage techniques. In *Proceedings of 21st IEEE/NASA Goddard MSS* (2004).

[37] YOU, L. L., POLLACK, K. T., AND LONG, D. D. E. Deep Store: An Archival Storage System Architecture. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering* (Washington, DC, USA, 2005), pp. 804–8015.

[38] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the disk bottleneck in the data domain deduplication file system. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2008), USENIX Association, pp. 1–14.

[39] ZHU, Y., JIANG, H., AND WANG, J. Hierarchical Bloom filter arrays (HBA): a novel, scalable metadata management system for large cluster-based storage. In *Cluster Computing, 2004 IEEE International Conference on* (Sept. 2004), pp. 165–174.

Evaluating Performance and Energy in File System Server Workloads

Priya Sehgal, Vasily Tarasov, and Erez Zadok
Stony Brook University

Abstract

Recently, power has emerged as a critical factor in designing components of storage systems, especially for power-hungry data centers. While there is some research into power-aware storage stack components, there are no systematic studies evaluating each component's impact separately. This paper evaluates the file system's impact on energy consumption and performance. We studied several popular Linux file systems, with various mount and format options, using the FileBench workload generator to emulate four server workloads: Web, database, mail, and file server. In case of a server node consisting of a single disk, CPU power generally exceeds disk-power consumption. However, file system design, implementation, and available features have a significant effect on CPU/disk utilization, and hence on performance and power. We discovered that default file system options are often suboptimal, and even poor. We show that a careful matching of expected workloads to file system types and options can improve power-performance efficiency by a factor ranging from 1.05 to 9.4 times.

1 Introduction

Performance has a long tradition in storage research. Recently, power consumption has become a growing concern. Recent studies show that the energy used inside all U.S. data centers is 1–2% of total U.S. energy consumption [42], with more spent by other IT infrastructures outside the data centers [44]. Storage stacks have grown more complex with the addition of virtualization layers (RAID, LVM), stackable drivers and file systems, virtual machines, and network-based storage and file system protocols. It is challenging today to understand the behavior of storage layers, especially when using complex applications.

Performance and energy use have a non-trivial, poorly understood relationship: sometimes they are opposites (e.g., spinning a disk faster costs more power but improves performance); but at other times they go hand in hand (e.g., localizing writes into adjacent sectors can improve performance while reducing the energy). Worse, the growing number of storage layers further perturb access patterns each time applications' requests traverse the layers, further obfuscating these relationships.

Traditional energy-saving techniques use *right-sizing*. These techniques adjust node's computational power to fit the current load. Examples include spinning disks down [12, 28, 30], reducing CPU frequencies and voltages [46], shutting down individual CPU cores, and putting entire machines into lower power states [13, 32]. Less work has been done on *workload-reduction* techniques: better algorithms and data-structures to improve power/performance [14, 19, 24]. A few efforts focused on energy-performance tradeoffs in parts of the storage stack [8, 18, 29]. However, they were limited to one problem domain or a specific workload scenario.

Many factors affect power and performance in the storage stack, especially workloads. Traditional file systems and I/O schedulers were designed for generality, which is ill-suited for today's specialized servers with long-running services (Web, database, email). We believe that to improve performance and reduce energy use, custom storage layers are needed for specialized workloads. But before that, thorough systematic studies are needed to recognize the features affecting power-performance under specific workloads.

This paper studies the impact of server workloads on both power and performance. We used the FileBench [16] workload generator due to its flexibility, accuracy, and ability to scale and stress any server. We selected FileBench's Web, database, email, and file server workloads as they represent most common server workloads, yet they differ from each other. Modern storage stacks consist of multiple layers. Each layer independently affects the performance and power consumption of a system, and together the layers make such interaction rather complex. Here, we focused on the file system layer only; to make this study a useful stepping stone towards understanding the entire storage stack, we did not use LVM, RAID, or virtualization. We experimented with Linux's four most popular and stable local file systems: Ext2, Ext3, XFS, and Reiserfs; and we varied several common format- and mount-time options to evaluate their impact on power/performance.

We ran many experiments on a server-class machine, collected detailed performance and power measurements, and analyzed them. We found that different workloads, not too surprisingly, have a large impact on system behavior. No single file system worked best for all workloads. Moreover, default file system format and mount options were often suboptimal. Some file system features helped power/performance and others hurt it. Our experiments revealed a strong linearity between the power efficiency and performance of a file system. Overall, we found significant variations in the amount of useful work that can be accomplished per unit time or unit energy, with possible improvements over default configurations ranging from 5% to 9.4×. We conclude that long-running servers should be carefully configured at installation time. For busy servers this can yield significant performance and power savings over time. We hope this study will inspire other studies (e.g., distributed file

systems), and lead to novel storage layer designs.

The rest of this paper is organized as follows. Section 2 surveys related work. Section 3 introduces our experimental methodology. Section 4 provides useful information about energy measurements. The bulk of our evaluation and analysis is in Section 5. We conclude in Section 6 and describe future directions in Section 7.

2 Related Work

Past power-conservation research for storage focused on portable battery-operated computers [12, 25]. Recently, researchers investigated data centers [9, 28, 43]. As our focus is file systems' power and performance, we discuss three areas of related work that mainly cover both power and performance: file system studies, lower-level storage studies, and benchmarks commonly used to evaluate systems' power efficiency.

File system studies. Disk-head seeks consume a large portion of hard-disk energy [2]. A popular approach to optimize file system power-performance is to localize on-disk data to incur fewer head movements. Huang et al. replicated data on disk and picked the closest replica to the head's position at runtime [19]. The Energy-Efficient File System (EEFS) groups files with high temporal access locality [24]. Essary and Amer developed predictive data grouping and replication schemes to reduce head movements [14].

Some suggested other file-system—level techniques to reduce power consumption without degrading performance. BlueFS is an energy-efficient distributed file system for mobile devices [29]. When applications request data, BlueFS chooses a replica that best optimizes energy and performance. GreenFS is a stackable file system that combines a remote network disk and a local flash-based memory buffer to keep the local disk idling for as long as possible [20]. Kothiyal et al. examined file compression to improve power and performance [23].

These studies propose new designs for storage software, which limit their applicability to existing systems. Also, they often focus on narrow problem domains. We, however, focus on servers, several common workloads, and use existing unmodified software.

Lower-level storage studies. A disk drive's platters usually keep spinning even if there are no incoming I/O requests. Turning the spindle motor off during idle periods can reduce disk energy use by 60% [28]. Several studies suggest ways to predict or prolong idle periods and shut the disk down appropriately [10, 12]. Unlike laptop and desktop systems, idle periods in server workloads are commonly too short, making such approaches ineffective. This was addressed using I/O off-loading [28], power-aware (sometimes flash-based) caches [5, 49], prefetching [26, 30], and a combination of these techniques [11, 43]. Massive Array of Idle Disks (MAID) augments RAID technology with automatic shut down of idle disks [9]. Pinheiro and Bianchini used the fact that regularly only a small subset of data is accessed by a system, and migrated frequently accessed data to a small number of active disks, keeping the remaining disks off [31]. Other approaches dynamically control the platters' rotation speed [35] or combine low- and high-speed disks [8].

These approaches depend primarily on having or prolonging idle periods, which is less likely on busy servers. For those, aggressive use of shutdown, slowdown, or spin-down techniques can have adverse effects on performance and energy use (e.g., disk spin-up is slow and costs energy); such aggressive techniques can also hurt hardware reliability. Whereas idle-time techniques are complementary to our study, we examine file systems' features that increase performance and reduce energy use in *active* systems.

Benchmarks and systematic studies. Researchers use a wide range of benchmarks to evaluate the performance of computer systems [39, 41] and file systems specifically [7, 16, 22, 40]. Far fewer benchmarks exist to determine system power efficiency. The Standard Performance Evaluation Corporation (SPEC) proposed the SPECpower_ssj benchmark to evaluate the energy efficiency of systems [38]. SPECpower_ssj stresses a Java server with standardized workload at different load levels. It combines results and reports the number of Java operations per second per watt. Rivoire et al. used a large sorting problem (guaranteed to exceed main memory) to evaluate a system's power efficiency [34]; they report the number of sorted records per joule. We use similar metrics, but applied for file systems.

Our goal was to conduct a systematic power-performance study of file systems. Gurumurthi et al. carried out a similar study for various RAID configurations [18], but focused on database workloads alone. They noted that tuning RAID parameters affected power and performance more than many traditional optimization techniques. We observed similar trends, but for file systems. In 2002, Bryant et al. evaluated Linux file system performance [6], focusing on scalability and concurrency. However, that study was conducted on an older Linux 2.4 system. As hardware and software change so rapidly, it is difficult to extrapolate from such older studies—another motivation for our study here.

3 Methodology

This section details the experimental hardware and software setup for our evaluations. We describe our testbed in Section 3.1. In Section 3.2 we describe our benchmarks and tools used. Sections 3.3 and 3.4 motivate our selection of workloads and file systems, respectively.

3.1 Experimental Setup

We conducted our experiments on a Dell PowerEdge SC1425 server consisting of 2 dual-core Intel® Xeon™ CPUs at 2.8GHz, 2GB RAM, and two 73GB internal SATA disks. The server was running the CentOS 5.3 Linux distribution with kernel 2.6.18-128.1.16.el5.centos.plus. All the benchmarks were executed on an external 18GB, 15K RPM ATLAS15K_18WLS Maxtor SCSI disk connected through Adaptec ASC-39320D Ultra320 SCSI Card.

As one of our goals was to evaluate file systems' impact on CPU and disk power consumption, we connected the machine and the external disk to two separate WattsUP Pro ES [45] power meters. This is an in-line power meter that measures the energy drawn by a device plugged into the meter's receptacle. The power meter uses non-volatile memory to store measurements every second. It has a 0.1 Watt-hour (1 Watt-hour = 3,600 Joules) resolution for energy measurements; the accuracy is $\pm 1.5\%$ of the measured value plus a constant error of ± 0.3 Watt-hours. We used a `wattsup` Linux utility to download the recorded data from the meter over a USB interface to the test machine. We kept the temperature in the server room constant.

3.2 Software Tools and Benchmarks

We used *FileBench* [16], an application level workload generator that allowed us to emulate a large variety of workloads. It was developed by Sun Microsystems and was used for performance analysis of Solaris operating system [27] and in other studies [1, 17]. FileBench can emulate different workloads thanks to its flexible *Workload Model Language* (WML), used to describe a workload. A WML workload description is called a *personality*. Personalities define one or more groups of file system operations (e.g., read, write, append, stat), to be executed by multiple threads. Each thread performs the group of operations repeatedly, over a configurable period of time. At the end of the run, FileBench reports the total number of performed operations. WML allows one to specify synchronization points between threads and the amount of memory used by each thread, to emulate real-world application more accurately. Personalities also describe the directory structure(s) typical for a specific workload: average file size, directory depth, the total number of files, and alpha parameters governing the file and directory sizes that are based on a gamma random distribution.

To emulate a real application accurately, one needs to collect system call traces of an application and convert them to a personality. FileBench includes several predefined personalities—Web, file, mail and database servers—which were created by analyzing the traces of corresponding applications in the enterprise environment [16]. We used these personalities in our study.

We used Auto-pilot [47] to drive FileBench. We built an Auto-pilot plug-in to communicate with the power meter and modified FileBench to clear the two watt meters' internal memory before each run. After each benchmark run, Auto-Pilot extracts the energy readings from both watt-meters. FileBench reports file system performance in operations per second, which Auto-pilot collects. We ran all tests at least five times and computed the 95% confidence intervals for the mean operations per second, and disk and CPU energy readings using the Student's-t distribution. Unless otherwise noted, the half widths of the intervals were less than 5% of the mean—shown as error bars in our bar graphs. To reduce the impact of the watt-meter's constant error (0.3 Watt-hours) we increased FileBench's default runtime from one to 10 minutes. Our test code, configuration files, logs, and results are available at `www.fsl.cs.sunysb.edu/docs/fsgreen-bench/`.

3.3 Workload Categories

One of our main goals was to evaluate the impact of different file system workloads on performance and power use. We selected four common server workloads: Web server, file server, mail server, and database server. The distinguishing workload features were: file size distributions, directory depths, read-write ratios, meta-data vs. data activity, and access patterns (i.e., sequential vs. random vs. append). Table 1 summarizes our workloads' properties, which we detail next.

Web Server. The Web server workload uses a read-write ratio of 10:1, and reads entire files sequentially by multiple threads, as if reading Web pages. All the threads append 16KB to a common Web log, thereby contending for that common resource. This workload not only exercises fast lookups and sequential reads of small-sized files, but it also considers concurrent data and meta-data updates into a single, growing Web log.

File Server. The file server workload emulates a server that hosts home directories of multiple users (threads). Users are assumed to access files and directories belonging only to their respective home directories. Each thread picks up a different set of files based on its thread id. Each thread performs a sequence of create, delete, append, read, write, and stat operations, exercising both the meta-data and data paths of the file system.

Mail Server. The mail server workload (varmail) emulates an electronic mail server, similar to Postmark [22], but it is multi-threaded. FileBench performs a sequence of operations to mimic reading mails (open, read whole file, and close), composing (open/create, append, close, and fsync) and deleting mails. Unlike the file server and Web server workloads, the mail server workload uses a

Workload	Average file size	Average directory depth	Number of files	I/O sizes			Number of threads	R/W Ratio
				read	write	append		
Web Server	32KB	3.3	20,000	1MB	-	16KB	100	10:1
File Server	256KB	3.6	50,000	1MB	1MB	16KB	100	1:2
Mail Server	16KB	0.8	50,000	1MB	-	16KB	100	1:1
DB Server	0.5GB	0.3	10	2KB	2KB	-	200 + 10	20:1

Table 1: FileBench workload characteristics. The database workload uses 200 readers and 10 writers.

flat directory structure, with all the files in one directory. This exercises large directory support and fast lookups. The average file size for this workload is 16KB, which is the smallest amongst all other workloads. This initial file size, however, grows later due to appends.

Database Server. This workload targets a specific class of systems, called *online transaction processing* (OLTP). OLTP databases handle real-time transaction-oriented applications (e.g., e-commerce). The database emulator performs random asynchronous writes, random synchronous reads, and moderate (256KB) synchronous writes to the log file. It launches 200 reader processes, 10 asynchronous writers, and a single log writer. This workload exercises large file management, extensive concurrency, and random reads/writes. This leads to frequent cache misses and on-disk file access, thereby exploring the storage stack's efficiency for caching, paging, and I/O.

3.4 File System and Properties

We ran our workloads on four different file systems: Ext2, Ext3, Reiserfs, and XFS. We evaluated both the default and variants of mount and format options for each file system. We selected these file systems for their widespread use on Linux servers and the variation in their features. Distinguishing file system features were:

- B+/S+ Tree vs. linear fixed sized data structures
- Fixed block size vs. variable-sized extent
- Different allocation strategies
- Different journal modes
- Other specialized features (e.g., tail packing)

For each file system, we tested the impact of various format and mount options that are believed to affect performance. We considered two common format options: block size and inode size. Large block sizes improve I/O performance of applications using large files due to fewer number of indirections, but they increase fragmentation for small files. We tested block sizes of 1KB, 2KB, and 4KB. We excluded 8KB block sizes due to lack of full support [15, 48]. Larger inodes can improve data locality by embedding as much data as possible inside the inode. For example, large enough inodes can hold small directory entries and small files directly, avoiding the need for disk block indirections. Moreover, larger inodes help storing the extent file maps. We tested the default (256B and 128B for XFS and Ext2/Ext3, respectively) and 1KB inode size for all file systems except Reiserfs, as it does not explicitly have an inode object.

We evaluated various mount options: `noatime`, journal vs. no journal, and different journalling modes. The `noatime` option improves performance in read-intensive workloads, as it skips updating an inode's last access time. Journalling provides reliability, but incurs an extra cost in logging information. Some file systems support different journalling modes: data, ordered, and writeback. The data journalling mode logs both data and meta-data. This is the safest but slowest mode. Ordered mode (default in Ext3 and Reiserfs) logs only meta-data, but ensures that data blocks are written before meta-data. The writeback mode logs meta-data without ordering data/meta-data writes. Ext3 and Reiserfs support all three modes, whereas XFS supports only the writeback mode. We also assessed a few file-system specific mount and format options, described next.

Ext2 and Ext3. Ext2 [4] and Ext3 [15] have been the default file systems on most Linux distributions for years. Ext2 divides the disk partition into fixed sized blocks, which are further grouped into similar-sized *block groups*. Each block group manages its own set of inodes, a free data block bitmap, and the actual files' data. The block groups can reduce file fragmentation and increase reference locality by keeping files in the same parent directory and their data in the same block group. The maximum block group size is constrained by the block size. Ext3 has an identical on-disk structure as Ext2, but adds journalling. Whereas journalling might degrade performance due to extra writes, we found certain cases where Ext3 outperforms Ext2. One of Ext2 and Ext3's major limitations is their poor scalability to large files and file systems because of the fixed number of inodes, fixed block sizes, and their simple array-indexing mechanism [6].

XFS. XFS [37] was designed for scalability: supporting terabyte sized files on 64-bit systems, an unlimited number of files, and large directories. XFS employs B+ trees to manage dynamic allocation of inodes, free space, and to map the data and meta-data of files/directories. XFS stores all data and meta-data in variable sized, contiguous *extents*. Further, XFS's partition is divided into fixed-sized regions called *allocation groups* (AGs), which are similar to block groups in Ext2/3, but are designed for scalability and parallelism. Each AG

manages the free space and inodes of its group independently; increasing the number of allocation groups scales up the number of parallel file system requests, but too many AGs also increases fragmentation. The default AG count value is 16. XFS creates a cluster of inodes in an AG as needed, thus not limiting the maximum number of files. XFS uses a delayed allocation policy that helps in getting large contiguous extents, and increases the performance of applications using large-sized files (e.g., databases). However, this increases memory utilization. XFS tracks AG free space using two B+ trees: the first B+ tree tracks free space by block number and the second tracks by the size of the free space block. XFS supports only meta-data journalling (writeback). Although XFS was designed for scalability, we evaluate all file systems using different file sizes and directory depths. Apart from evaluating XFS's common format and mount options, we also varied its AG count.

Reiserfs. The Reiserfs partition is divided into blocks of fixed size. Reiserfs uses a *balanced S+ tree* [33] to optimize lookups, reference locality, and space-efficient packing. The S+ tree consists of internal nodes, formatted leaf nodes, and unformatted nodes. Each internal node consists of key-pointer pairs to its children. The formatted nodes pack objects tightly, called *items*; each item is referenced through a unique key (akin to an inode number). These items include: *stat items* (file meta-data), *directory items* (directory entries), *indirect items* (similar to inode block lists), and *direct items* (tails of files less than 4K). A formatted node accommodates items of different files and directories. Unformatted nodes contain raw data and do not assist in tree lookup. The direct items and the pointers inside indirect items point to these unformatted nodes. The internal and formatted nodes are sorted according to their keys. As a file's meta-data and data is searched through the combined S+ tree using keys, Reiserfs scales well for a large and deep file system hierarchy. Reiserfs has a unique feature we evaluated called *tail packing*, intended to reduce internal fragmentation and optimize the I/O performance of small sized files (less than 4K). Tail-packing support is enabled by default, and groups different files in the same node. These are referenced using direct pointers, called the tail of the file. Although the tail option looks attractive in terms of space efficiency and performance, it incurs an extra cost during reads if the tail is spread across different nodes. Similarly, additional appends to existing tail objects lead to unnecessary copy and movement of the tail data, hurting performance. We evaluated all three journalling modes of Reiserfs.

4 Energy Breakdown

Active vs. passive energy. Even when a server does not perform any work, it consumes some energy. We call this energy *idle* or *passive*. The file system selection alone cannot reduce idle power, but combined with right-sizing techniques, it can improve power efficiency by prolonging idle periods. The *active* power of a node is an additional power drawn by the system when it performs useful work. Different file systems exercise the system's resources differently, directly affecting active power. Although file systems affect active energy only, users often care about total energy used. Therefore, we report only total power used.

Hard disk vs. node power. We collected power consumption readings for the external disk drive and the test node separately. We measured our hard disk's idle power to be 7 watts, matching its specification. We wrote a tool that constantly performs direct I/O to distant disk tracks to maximize its power consumption, and measured a maximum power of 22 watts. However, the average disk power consumed for our experiments was only 14 watts with little variations. This is because the workloads exhibited high locality, heavy CPU/memory use, and many I/O requests were satisfied from caches. Whenever the workloads did exercise the disk, its power consumption was still small relative to the total power. Therefore, for the rest of this paper, we report only total system power consumption (disk included).

A node's power consumption consists of its components' power. Our server's measured idle-to-peak power is 214–279W. The CPU tends to be a major contributor, in our case from 86–165W (i.e., Intel's SpeedStep technology). However, the behavior of power consumption within a computer is complex due to thermal effects and feedback loops. For example, our CPU's core power use can drop to a mere 27W if its temperature is cooled to 50 °C, whereas it consumes 165W at a normal temperature of 76 °C. Motherboards today include dynamic system and CPU fans which turn on/off or change their speeds; while they reduce power elsewhere, the fans consume some power themselves. For simplicity, our paper reports only total system power consumption.

FS vs. other software power consumption. It is reasonable to question how much energy does a file system consume compared to other software components. According to Almeida et al., a Web server saturated by client requests spends 90% of the time in kernel space, invoking mostly file system related system calls [3]. In general, if a user-space program is not computationally intensive, it frequently invokes system calls and spends a lot of time in kernel space. Therefore, it makes sense to focus the efforts on analyzing energy efficiency of file systems. Moreover, our results in Section 5 support this fact: changing only the file system type can increase power/performance numbers up to a factor of 9.

5 Evaluation

This section details our results and analysis. We abbreviated the terms Ext2, Ext3, Reiserfs, and XFS as `e2`, `e3`, `r`, and `x`, respectively. File systems formatted with block size of 1K and 2K are denoted `blk1k` and `blk2k`, respectively; `isz1k` denotes 1K inode sizes; `bg16k` denotes 16K block group sizes; `dtlg` and `wrbck` denote data and writeback journal modes, respectively; `nolog` denotes Reiserfs's no-logging feature; allocation group count is abbreviated as `agc` followed by number of groups (8, 32, etc.), no-atime is denoted as `noatm`.

Section 5.1 overviews our metrics and terms. We detail the Web, File, Mail, and DB workload results in Sections 5.2–5.5. Section 5.6 provides recommendations for selecting and designing efficient file systems.

5.1 Overview

In all our tests, we collected two raw metrics: performance (from FileBench), and the average power of the machine and disk (from watt-meters). FileBench reports file system performance under different workloads in units of *operations per second* (ops/sec). As each workload targets a different application domain, this metric is not comparable across workloads: A Web server's ops/sec are not the same as, say, the database server's. Their magnitude also varies: the Web server's rates numbers are two orders of magnitude larger than other workloads. Therefore, we report Web server performance in 1,000 ops/sec, and just ops/sec for the rest.

Electrical power, measured in Watts, is defined as the rate at which electrical energy is transferred by a circuit. Instead of reporting the raw power numbers, we selected a derived metric called *operations per joule* (ops/joule), which better explains power efficiency. This is defined as the amount of work a file system can accomplish in 1 Joule of energy ($1Joule = 1watt \times 1sec$). The higher the value, the more power-efficient the system is. This metric is similar to SPEC's ($\frac{ssj\_ops}{watt}$) metric, used by SPECPower_ssj2008 [38]. Note that we report the Web server's power efficiency in ops/joule, and use ops/kilojoule for the rest.

A system's active power consumption depends on how much it is being utilized by software, in our case a file system. We measured that the higher the system/CPU utilization, the greater the power consumption. We therefore ran experiments to measure the power consumption of a workload at different load levels (i.e., ops/sec), for all four file systems, with default format and mount options. Figure 1 shows the average power consumed (in Watts) by each file system, increasing Web server loads from 3,000 to 70,000 ops/sec. We found that all file systems consumed almost the same amount of energy at a certain performance levels, but only a few could withstand more load than the others. For example,

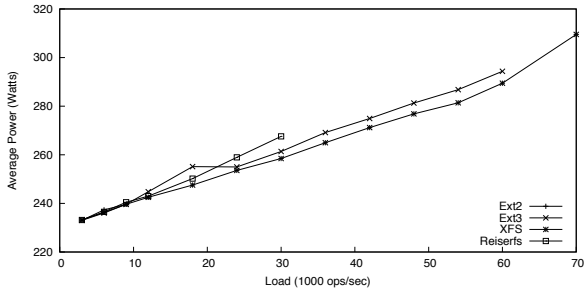


Figure 1: Webserver: Mean power consumption by Ext2, Ext3, Reiserfs, and XFS at different load levels. The y-axis scale starts at 220 Watts. Ext2 does not scale above 10,000 ops/sec.

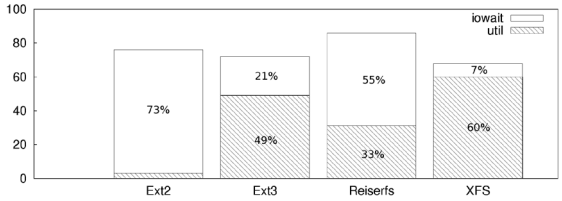
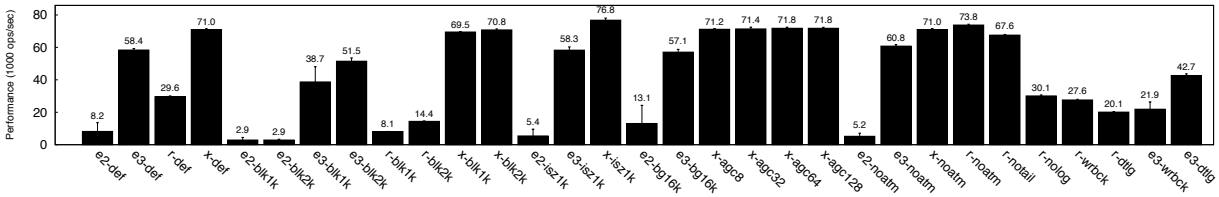


Figure 2: Average CPU utilization for the Webserver workload

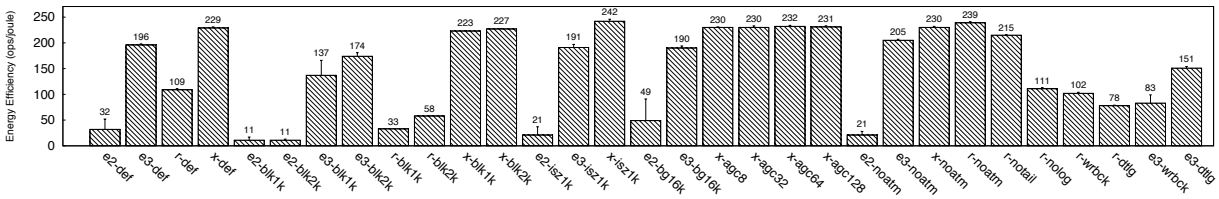
Ext2 had a maximum of only 8,160 Web ops/sec with an average power consumption of 239W, while XFS peaked at 70,992 ops/sec, with only 29% more power consumption. Figure 2 shows the percentages of CPU utilization, I/O wait, and idle time for each file system at its maximum load. Ext2 and Reiserfs spend more time waiting for I/O than any other file system, thereby performing less useful work, as per Figure 1. XFS consumes almost the same amount of energy as the other three file systems at lower load levels, but it handles much higher Web server loads, winning over others in both power efficiency and performance. We observed similar trends for other workloads: only one file system outperformed the rest in terms of both power and performance, at all load levels. Thus, in the rest of this paper we report only peak performance figures.

5.2 Webserver Workload

As we see in Figures 3(a) and 3(b), XFS proved to be the most power- and performance-efficient file system. XFS performed 9 times better than Ext2, as well as 2 times better than Reiserfs, in terms of both power and performance. Ext3 lagged behind XFS by 22%. XFS wins over all the other file systems as it handles concurrent updates to a single file efficiently, without incurring a lot of I/O wait (Figure 2), thanks to its journal design. XFS maintains an active item list, which it uses to prevent meta-data buffers from being written multiple times if they belong to multiple transactions. XFS pins a meta-data buffer to prevent it from being written to the disk until the log is committed. As XFS batches multiple updates to a common inode together, it utilizes the CPU better. We observed a linear relationship between power-efficiency and performance for the Web server workload,



(a) File system Webserver workload performance (in 1000 ops/sec)



(b) File system energy efficiency for Webserver workload (in ops/joule)

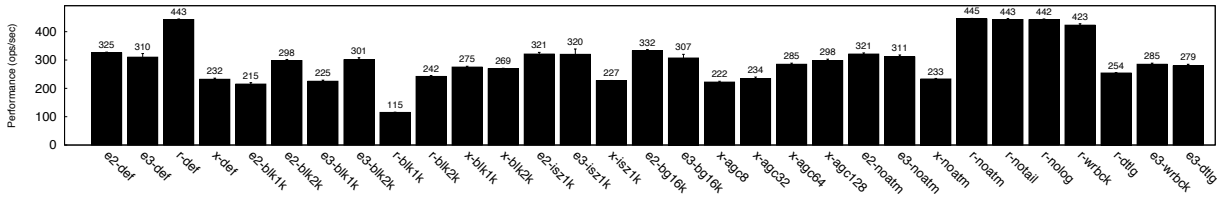
Figure 3: File system performance and energy efficiency under the Webserver workload

so we report below on the basis of performance alone.

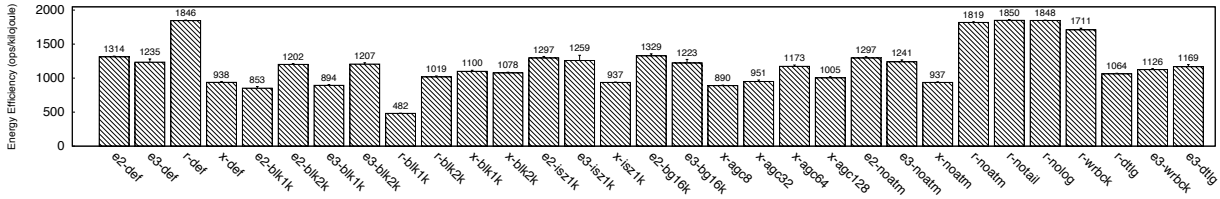
Ext2 performed the worst and exhibited inconsistent behavior. Its standard deviation was as high as 80%, even after 30 runs. We plotted the performance values on a histogram and observed that Ext2 had a non-Gaussian (long-tailed) distribution. Out of 30 runs, 21 runs (70%) consumed less than 25% of the CPU, while the remaining ones used up to 50%, 75%, and 100% of the CPU (three runs in each bucket). We wrote a micro-benchmark which ran for a fixed time period and appended to 3 common files shared between 100 threads. We found that Ext3 performed 13% fewer appends than XFS, while Ext2 was 2.5 times slower than XFS. We then ran a modified Web server workload with *only* reads and no log appends. In this case, Ext2 and Ext3 performed the same, with XFS lagging behind by 11%. This is because XFS's `lookup` operation takes more time than other file systems for deeper hierarchy (see Section 5.3). As XFS handles concurrent writes better than the others, it overcomes the performance degradation due to slow lookups and outperforms in the Web server workload. OSprof results [21] revealed that the average latency of `write_super` for Ext2 was 6 times larger than Ext3. Analyzing the file systems' source code helped explain this inconsistency. First, as Ext2 does not have a journal, it commits superblock and inode changes to the on-disk image immediately, without batching changes. Second, Ext2 takes the global kernel lock (aka BKL) while calling `ext2_write_super` and `ext2_write_inode`, which further reduce parallelism: all processes using Ext2 which try to sync an inode or the superblock to disk will contend with each other, increasing wait times significantly. On the contrary, Ext3 batches all updates to the inodes in the journal and only when the JBD layer calls `journal_commit_transaction` are all the metadata updates actually synced to the disk (after committing the data). Although journalling was designed primarily for reliability reasons, we conclude that a careful journal design can help some concurrent-write workloads akin to LFS [36].

Reiserfs exhibits poor performance for different reasons than Ext2 and Ext3. As Figures 3(a) and 3(b) show, Reiserfs (default) performed worse than both XFS and Ext3, but Reiserfs with the `notail` mount option outperformed Ext3 by 15% and the default Reiserfs by 2.25 times. The reason is that by default the `tail` option is enabled in Reiserfs, which tries to pack all files less than 4KB in one block. As the Web server has an average file size of just 32KB, it has many files smaller than 4KB. We confirmed this by running `debugreiserfs` on the Reiserfs partition: it showed that many small files had their data spread across the different blocks (packed along with other files' data). This resulted in more than one data block access for each file read, thereby increasing I/O, as seen in Figure 2. We concluded that unlike Ext2 and Ext3, the default Reiserfs experienced a performance hit due to its small file read design, rather than concurrent appends. This demonstrates that even simple Web server workload can still exercise different parts of file systems' code.

An interesting observation was that the `noatime` mount option improved the performance of Reiserfs by a factor of 2.5 times. In other file systems, this option did not have such a significant impact. The reason is that the `reiserfs_dirty_inode` function, which updates the access time field, acquires the BKL and then searches for the stat item corresponding to the inode in its S+ tree to update the `atime`. As the BKL is held while updating each inode's access time in a path, it hurts parallelism and reduces performance significantly. Also, `noatime` boosts Reiserfs's performance by this



(a) Performance of file systems for the file server workload (in ops/sec)



(b) Energy efficiency of file systems for the file server workload (in ops/kilojoule)

Figure 4: Performance and energy efficiency of file systems under the file server workload

much *only* in the read-intensive Web server workload.

Reducing the block-size during format generally hurt performance, except in XFS. XFS was unaffected thanks to its delayed allocation policy that allocates a large contiguous extent, irrespective of the block size; this suggests that modern file systems should try to pre-allocate large contiguous extents in anticipation of files' growth. Reiserfs observed a drastic degradation of 2–3× after decreasing the block size from 4KB (default) to 2KB and 1KB, respectively. We found from `debugreiserfs` that this led to an increase in the number of internal and formatted nodes used to manage the file system namespace and objects. Also, the height of the S+ tree grew from 4 to 5, in case of 1KB. As the internal and formatted nodes depend on the block size, a smaller block size reduces the number of entries packed inside each of these nodes, thereby increasing the number of nodes, and increasing I/O times to fetch these nodes from the disk during lookup. Ext2 and Ext3 saw a degradation of 2× and 12%, respectively, because of the extra indirections needed to reference a single file. Note that Ext2's 2× degradation was coupled with a high standard variation of 20–49%, for the same reasons explained above.

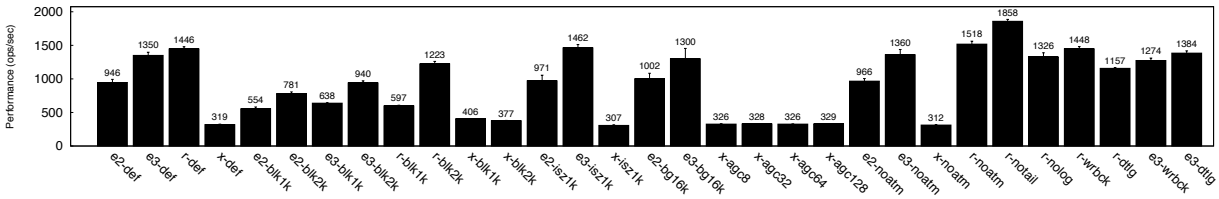
Quadrupling the XFS inode size from 256B to 1KB improved performance by only 8%. We found using `xfs_db` that a large inode allowed XFS to embed more extent information and directory entries inside the inode itself, speeding lookups. As expected, the data journalling mode hurt performance for both Reiserfs and Ext3 by 32% and 27%, respectively. The writeback journalling mode of Ext3 and Reiserfs degraded performance by 2× and 7%, respectively, compared to their default ordered journalling mode. Increasing the block group count of Ext3 and the allocation group count of XFS had a negligible impact. The reason is that the Web server is a read-intensive workload, and does not need to update the different group's metadata as frequently as a write-intensive workload would.

5.3 File Server Workload

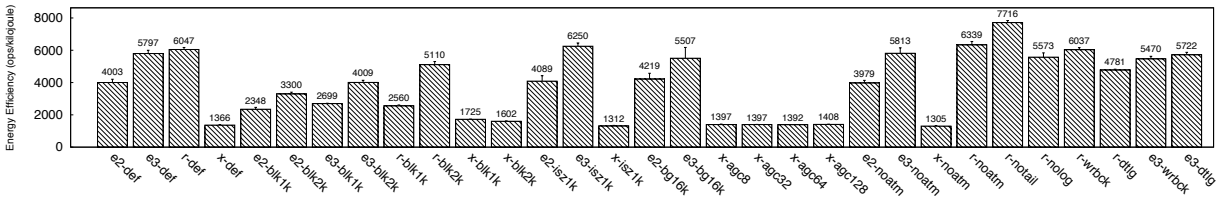
Figures 4(a) and 4(b) show that Reiserfs outperformed Ext2, Ext3, XFS by 37%, 43%, and 91%, respectively. Compared to the Web server workload, Reiserfs performed better than all others, even with the `tail` option on. This is because the file server workload has an average file size of 256KB (8 times larger than the Web server workload): it does not have many small files spread across different nodes, thereby showing no difference between Reiserfs's (`tail`) and `no-tail` options.

Analyzing using OSprof revealed that XFS consumed 14% and 12% more time in `lookup` and `create`, respectively, than Reiserfs. Ext2 and Ext3 spent 6% more time in both `lookup` and `create` than Reiserfs. To exercise only the lookup path, we executed a simple micro-benchmark that only performed open and close operations on 50,000 files by 100 threads, and we used the same fileset parameters as that of the file server workload (see Table 1). We found that XFS performed 5% fewer operations than Reiserfs, while Ext2 and Ext3 performed close to Reiserfs. As Reiserfs packs data and meta-data all in one node and maintains a balanced tree, it has faster lookups thanks to improved spatial locality. Moreover, Reiserfs stores objects by sorted keys, further speeding lookup times. Although XFS uses B+ trees to maintain its file system objects, its spatial locality is worse than that of Reiserfs, as XFS has to perform more hops between tree nodes.

Unlike the Web server results, Ext2 performed better than Ext3, and did not show high standard deviations. This was because in a file server workload, each thread works on an independent set of files, with little contention to update a common inode.



(a) Performance of file systems under the varmail workload (in ops/sec)



(b) Energy efficiency of file systems under the varmail workload (in ops/kilojoule)

Figure 5: Performance and energy efficiency of file systems under the varmail workload

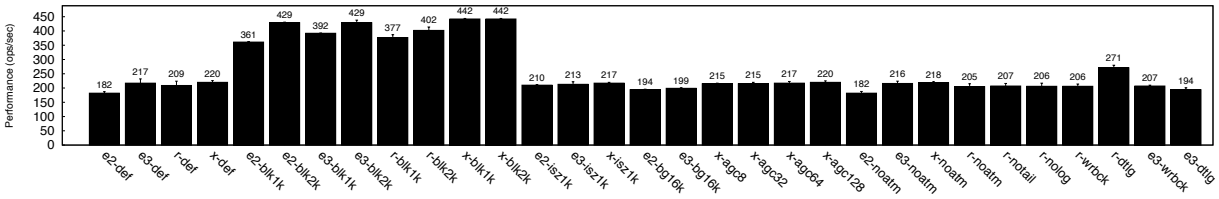
We discovered an interesting result when varying XFS's allocation group (AG) count from 8 to 128, in powers of two (default is 16). XFS's performance increased from 4% to 34% (compared to AG of 8). But, XFS's power efficiency increased linearly only until the AG count hit 64, after which the ops/kilojoule count dropped by 14% (for AG count of 128). Therefore, XFS' AG count exhibited a *non-linear* relationship between power-efficiency and performance. As the number of AGs increases, XFS's parallelism improves too, boosting performance even when dirtying each AG at a faster rate. However, all AGs share a common journal: as the number of AGs increases, updating the AG descriptors in the log becomes a bottleneck; we see diminishing returns beyond AG count of 64. Another interesting observation is that AG count increases had a negligible effect of only 1% improvement for the Web server, but a significant impact in file server workload. This is because the file server has a greater number of meta-data activities and writes than the Web server (see Section 3), thereby accessing/modifying the AG descriptors frequently. We conclude that the AG count is sensitive to the workload, especially read-write and meta-data update ratios. Lastly, the block group count increase in Ext2 and Ext3 had a small impact of less than 1%.

Reducing the block size from 4KB to 2KB improved the performance of XFS by 16%, while a further reduction to 1KB improved the performance by 18%. Ext2, Ext3, and Reiserfs saw a drop in performance, for the reasons explained in Section 5.2. Ext2 and Ext3 experienced a performance drop of 8% and 3%, respectively, when going from 4KB to 2KB; reducing the block size from 2KB to 1KB degraded their performance further by 34% and 27%, respectively. Reiserfs's performance declined by a 45% and 75% when we reduced the block size to 2KB and 1KB, respectively. This is due to the increased number of internal node lookups, which increase disk I/O as discussed in Section 5.2.

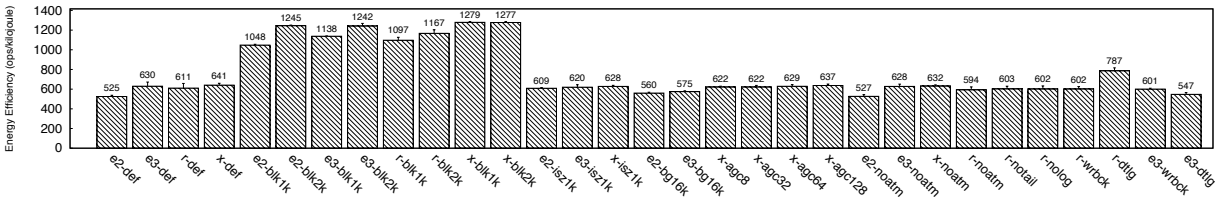
The `no-atime` options did not affect performance or power efficiency of any file system because this workload is not read-intensive and had a ratio of two writes for each read. Changing the inode size did not have an effect on Ext2, Ext3, or XFS. As expected, data journalling reduced the performance of Ext3 and Reiserfs by 10% and 43%, respectively. Writeback-mode journalling also showed a performance reduction by 8% and 4% for Ext3 and Reiserfs, respectively.

5.4 Mail Server

As seen in Figures 5(a) and 5(b), Reiserfs performed the best amongst all, followed by Ext3 which differed by 7%. Reiserfs beats Ext2 and XFS by 43% and 4×, respectively. Although the mail server's personality in FileBench is similar to the file server's, we observed differences in their results, because the mail server workload calls `fsync` after each append, which is not invoked in the file server workload. The `fsync` operation hurts the non-journalling version of file systems: hurting Ext2 by 30% and Reiserfs-nolog by 8% as compared to Ext3 and default Reiserfs, respectively. We confirmed this by running a micro-benchmark in FileBench which created the same directory structure as the mail server workload and performed the following sequence of operations: create, append, fsync, open, append, and fsync. This showed that Ext2 was 29% slower than Ext3. When we repeated this after removing all fsync calls, Ext2 and Ext3 performed the same. Ext2's poor performance with fsync calls is because its `ext2_sync_file` call ultimately invokes `ext2_write_inode`, which exhibits a larger latency than the `write_inode` function of other file systems. XFS's poor performance was due to its slower `lookup` operations.



(a) Performance of file systems for the OLTP workload (in ops/sec)



(b) Energy efficiency of file systems for the OLTP workload (in ops/kilojoule)

Figure 6: Performance and energy efficiency of file systems for the OLTP workload

Figure 5(a) shows that Reiserfs with `no-tail` beats all the variants of mount and format options, improving over default Reiserfs by 29%. As the average file size here was 16KB, the `no-tail` option boosted the performance similar to the Web server workload.

As in the Web server workload, when the block size was reduced from 4KB to 1KB, the performance of Ext2 and Ext3 dropped by 41% and 53%, respectively. Reiserfs's performance dropped by 59% and 15% for 1KB and 2KB, respectively. Although the performance of Reiserfs decreased upon reducing the block size, the percentage degradation was less than seen in the Web and file server. The flat hierarchy of the mail server attributed to this reduction in degradation; as all files resided in one large directory, the spatial locality of the meta data of these files increases, helping performance a bit even with smaller block sizes. Similar to the file server workload, reduction in block size increased the overall performance of XFS.

XFS's allocation group (AG) count and the block group count of Ext2 and Ext3 had minimal effect within the confidence interval. Similarly, the `no-atime` option and inode size did not impact the efficiency of file server significantly. The data journalling mode decreased Reiserfs's performance by 20%, but had a minimal effect on Ext3. Finally, the writeback journal mode decreased Ext3's performance by 6%.

5.5 Database Server Workload (OLTP)

Figures 6(a) and 6(b) show that all four file systems perform equally well in terms of both performance and power-efficiency with the default mount/format options, except for Ext2. It experiences a performance degradation of about 20% as compared to XFS. As explained in Section 5.2, Ext2's lack of a journal makes its random write performance worse than any other journalled file system, as they batch inode updates.

In contrast to other workloads, the performance of *all* file systems increases by a factor of around 2× if we decrease the block size of the file system from the default 4KB to 2KB. This is because the 2KB block size better matches the I/O size of OLTP workload (see Table 1), so every OLTP write request fits perfectly into the file system's block size. But, a file-system block size of 4KB turns a 2KB write into a read-modify-write sequence, requiring an extra read per I/O request. This proves an important point that keeping the file system block size close to the workload's I/O size can impact the efficiency of the system significantly. OLTP's performance also increased when using a 1KB block size, but was slightly lower than that obtained by 2KB block size, due to an increased number of I/O requests.

An interesting observation was that on decreasing the number of blocks per group from 32KB (default) to 16KB, Ext2's performance improved by 7%. Moreover, increasing the inode size up to 1KB improved performance by 15% as compared to the default configuration. Enlarging the inode size in Ext2 has an indirect effect on the blocks per group: the larger the inode size, the fewer the number of blocks per group. A 1KB inode size resulted in 8KB blocks per group, thereby doubling the number of block groups and increasing the performance as compared to the `e2-bg16K` case. Varying the AG count had a negligible effect on XFS's numbers. Unlike Ext2, the inode size increase did not affect any other file system.

Interestingly, we observed that the performance of Reiserfs increased by 30% on switching from the default ordered mode to the data journalling mode. In data journalling mode as all the data is first written to the log, random writes become logically sequential and achieve better performance than the other journalling modes.

FS	Option		Webserver		Fileserver		Varmail		Database	
	Type	Name	Perf.	Pow.	Perf.	Pow.	Perf.	Pow.	Perf.	Pow.
Ext2	mount	noatime	-37% †	-35%	-	-	-	-	-	-
	format	blk1k	-64% †	-65%	-34%	-35%	-41%	-41%	**+98%**	**+100%**
		blk2k	-65%	-65%	-8%	-9%	-17%	-18%	**+136%**	**+137%**
		isz1k	-34% †	-35%	-	-	-	-	**+15%**	**+16%**
		bg16k	**+60%** †	**+53%**	-	-	**+6%**	**+5%**	+7%	+7%
Ext3	mount	noatime	**+4%**	**+5%**	-	-	-	-	-	-
		dtlg	-27%	-23%	-10%	-5%	-	-	-11%	-13%
		wrbck	-63%	-57%	-8%	-9%	-6%	-5%	-5%	-5%
	format	blk1k	-34%	-30%	-27%	-28%	-53%	-53%	**+81%**	**+81%**
		blk2k	-12%	-11%	-	-	-30%	-31%	**+98%**	**+97%**
		isz1k	-	-	-	-	**+8%**	**+8%**	-	-
		bg16k	-	-	-	-	-4%	-5%	-8%	-9%
Reiserfs	mount	noatime	**+149%**	**+119%**	-	-	**+5%**	**+5%**	-	-
		notail	**+128%**	**+96%**	-	-	**+29%**	**+28%**	-	-
		nolog	-	-	-	-	-8%	-8%	-	-
		wrbck	-7%	-7%	-4%	-7%	-	-	-	-
		dtlg	-32%	-29%	-43%	-42%	-20%	-21%	**+30%**	**+29%**
	format	blk1k	-73%	-70%	-74%	-74%	-59%	-58%	**+80%**	**+80%**
		blk2k	-51%	-47%	-45%	-45%	-15%	-16%	**+92%**	**+91%**
XFS	mount	noatime	-	-	-	-	-	-	-	-
	format	blk1k	-	-	**+18%**	**+17%**	**+27%**	**+17%**	**+101%**	**+100%**
		blk2k	-	-	**+16%**	**+15%**	**+18%**	**+17%**	**+101%**	**+99%**
		isz1k	**+8%**	**+6%**	-	-	-	-	-	-
		agcnt8	-	-	-4%	-5%	-	-	-	-
		agcnt32	-	-	-	-	-	-	-	-
		agcnt64	-	-	**+23%**	**+25%**	-	-	-	-
		agcnt128	-	-	**+29%**	**+8%**	-	-	-	-

Table 2: File systems' performance and power, varying options, relative to the default ones for each file system. Improvements are highlighted in bold. A † denotes the results with coefficient of variation over 40%. A dash signifies statistically indistinguishable results.

In contrast to the Web server workload, the `no-atime` option does not have any effect on the performance of Reiserfs, although the read-write ratio is 20:1. This is because the database workload consists of only 10 large files and hence the meta-data of these small number of files (i.e., stat items) accommodate in a few formatted nodes as compared to the Web server workload which consists of 20,000 files with their meta-data scattered across multiple formatted nodes. Reiserfs' `no-tail` option had no effect on the OLTP workload due to the large size of its files.

5.6 Summary and Recommendations

We now summarize the combined results of our study. We then offer advice to server operators, as well as designers of future systems.

Staying within a file system type. Switching to a different file system type can be a difficult decision, especially in enterprise environments where policies may require using specific file systems or demand extensive testing before changing one. Table 2 compares the power efficiency and performance numbers that can be achieved while staying within a file system; each cell is a percentage of improvement (plus sign and bold font), or degradation (minus sign) compared to the *default* format and mount options for that file system. Dashes denote results that were statistically indistinguishable from default. We compare to the default case because file systems are often configured with default options.

Format and mount options represent different levels of optimization complexity. Remounting a file system with new options is usually seamless, while reformatting existing file systems requires costly data migration. Thus, we group mount and format options together.

From Table 2 we conclude that often there is a better selection of parameters than the default ones. A careful choice of file system parameters cuts energy use in half and more than doubles the performance (Reiserfs with `no-tail` option). On the other hand, a careless selection of parameters may lead to serious degradations: up to 64% drop in both energy and performance (e.g., legacy Ext2 file systems with 1K block size). Until October 1999, *mkfs.ext2* used 1KB block sizes by default.

File systems formatted prior to the time that Linux vendors picked up this change, still use small block sizes: performance-power numbers of a Web-server running on top of such a file system are 65% lower than today's default and over 4 times worse than best possible.

Given Table 2, we feel that even moderate improvements are worth a costly file system reformatting, because the savings accumulate for long-running servers.

Selecting the most suitable file system. When users can change to any file system, or choose one initially, we offer Table 3. For each workload we present the most power-performance efficient file system and its parameters. We also show the range of improvements in both ops/sec and ops/joule as compared to the best and worst *default* file systems. From the table we conclude that it is often possible to improve the efficiency by at least 8%. For the file server workload, where the default Reiserfs configuration performs the best, we observe a performance boost of up to 2× as compared to the worst default file system (XFS). As seen in Figure 5, for mail server workload Reiserfs with `no-tail` improves the efficiency by 30% over default Reiserfs (best default), and by 5× over default XFS (worst default). For the database workload, XFS with a block size of 2KB improved the efficiency of the system by at least two-fold. Whereas in most cases, performance and energy improved by nearly the same factor, in XFS they did not: for the Webserver workload, XFS with 1K inode sizes increased performance by a factor of 9.4 and energy improved by a factor of 7.5.

Some file system parameters listed in Table 2 can be combined, possibly yielding cumulative improvements. We analyzed several such combinations and concluded that each case requires careful investigation. For example, Reiserfs's `notail` and `noatime` options, independently, improved the Webserver's performance by 149% and 128%, respectively; but their combined effect only improved performance by 155%. The reason for this was that both parameters affected the same performance component—wait time—either by reducing BKL contention slightly or by reducing I/O wait time. However, the CPU's utilization remained high and dominated overall performance. On the other hand, XFS's `blk2k` and `agcnt64` format options, which improved performance by 18% and 23%, respectively—combined together to yield a cumulative improvement of 41%. The reason here is that these were options which affected different code paths without having other limiting factors.

Selecting file system features for a workload. We offer recommendations to assist in selecting the best file system feature(s) for specific workloads. These guideline can also help future file system designers.

Server	Recom. FS	Ops/Sec	Ops/Joule
Web	x-isz1k	1.08–9.4×	1.06–7.5×
File	r-def	1.0–1.9×	1.0–2.0×
Mail	r-notail	1.3–5.8×	1.3–5.7×
DB	x-blk2k	2–2.4×	2–2.4×

Table 3: Recommended file systems and their parameters for our workloads. We provide the range of performance and power-efficiency improvements achieved compared to the best and the worst default configured file systems.

- **File size:** If the workload generates or uses files with an average file size of a few 100KB, we recommend to use fixed sized data blocks, addressed by a balanced tree (e.g., Reiserfs). Large sized files (GB, TB) would benefit from extent-based balanced trees with delayed allocation (e.g., XFS). Packing small files together in one block (e.g., Reiserfs's tail-packing) is not recommended, as it often degrades performance.
- **Directory depth:** Workloads using a deep directory structure should focus on faster lookups using intelligent data structures and mechanisms. One recommendation is to localize as much data together with inodes and directories, embedding data into large inodes (XFS). Another is to sort all inodes/names and provide efficient balanced trees (e.g., XFS or Reiserfs).
- **Access pattern and parallelism:** If the workload has a mix of read, write, and metadata operations, it is recommended to use at least 64 allocation groups, each managing their own group and free data allocation independently, to increase parallelism (e.g., XFS). For workloads having multiple concurrent writes to the same file(s), we recommend to switch on journalling, so that updates to the same file system objects can be batched together. We recommend turning off `atime` updates for read-intensive operations, if the workload does not care about access-times.

6 Conclusions

Proper benchmarking and analysis are tedious, time-consuming tasks. Yet their results can be invaluable for years to come. We conducted a comprehensive study of file systems on modern systems, evaluated popular server workloads, and varied many parameters. We collected and analyzed performance and power metrics.

We discovered and explained significant variations in both performance and energy use. We found that there are no universally good configurations for all workloads, and we explained complex behavior that go against common conventions. We concluded that default file system types and options are often suboptimal: simple changes within a file system, like mount options, can improve power/performance from 5% to 149%; and chang-

ing format options can boost the efficiency from 6% to 136%. Switching to a different file system can result in improvements ranging from 2 to 9 times.

We recommend that servers be tested and optimized for expected workloads before used in production. Energy technologies lag far behind computing speed improvements. Given the long-running nature of busy Internet servers, software-based optimization techniques can have significant, cumulative long-term benefits.

7 Future Work

We plan to expand our study to include less mature file systems (e.g., Ext4, Reiser4, and BTRFS), as we believe they have greater optimization opportunities. We are currently evaluating power-performance of network-based and distributed file systems (e.g., NFS, CIFS, and Lustre). Those represent additional complexity: protocol design, client vs. server implementations, and network software and hardware efficiency. Early experiments comparing NFSv4 client/server OS implementations revealed performance variations as high as 3×.

Computer hardware changes constantly—e.g., adding more cores, and supporting more energy-saving features. As energy consumption outside of the data center exceeds that inside [44], we are continually repeating our studies on a range of computers spanning several years of age. We also plan to conduct a similar study on faster solid-state disks, and machines with more advanced DVFS support.

Our long-term goal is to develop custom file systems that best match a given workload. This could be beneficial because many application designers and administrators know their data set and access patterns ahead of time, allowing storage stacks designs with better cache behavior and minimal I/O latencies.

Acknowledgments. We thank the anonymous Usenix FAST reviewers and our shepherd, Steve Schlosser, for their helpful comments. We would also like to thank Richard Spillane, Sujay Godbole, and Saumitra Bhanage for their help. This work was made possible in part thanks to NSF awards CCF-0621463 and CCF-0937854, an IBM Faculty award, and a NetApp gift.

References

[1] A. Ermolinskiy and R. Tewari. C2Cfs: A Collective Caching Architecture for Distributed File Access. Technical Report UCB/EECS-2009-40, University of California, Berkeley, 2009.

[2] M. Allalouf, Y. Arbitman, M. Factor, R. I. Kat, K. Meth, and D. Naor. Storage Modeling for Power Estimation. In *Proceedings of the Israeli Experimental Systems Conference (SYSTOR '09)*, Haifa, Israel, May 2009. ACM.

[3] J. Almeida, V. Almeida, and D. Yates. Measuring the Behavior of a World-Wide Web Server. Technical report, Boston University, Boston, MA, USA, 1996.

[4] R. Appleton. A Non-Technical Look Inside the Ext2 File System. *Linux Journal*, August 1997.

[5] T. Bisson, S.A. Brandt, and D.D.E. Long. A Hybrid Disk-Aware Spin-Down Algorithm with I/O Subsystem Support. In *IEEE 2007 Performance, Computing, and Communications Conference*, 2007.

[6] R. Bryant, R. Forester, and J. Hawkes. Filesystem Performance and Scalability in Linux 2.4.17. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 259–274, Monterey, CA, June 2002. USENIX Association.

[7] D. Capps. IOzone Filesystem Benchmark. `www.iozone.org/`, July 2008.

[8] E. Carrera, E. Pinheiro, and R. Bianchini. Conserving Disk Energy in Network Servers. In *17th International Conference on Supercomputing*, 2003.

[9] 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.

[10] M. Craven and A. Amer. Predictive Reduction of Power and Latency (PuRPLe). In *Proceedings of the 22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST'05)*, pages 237–244, Washington, DC, USA, 2005. IEEE Computer Society.

[11] Y. Deng and F. Helian. EED: Energy Efficient Disk Drive Architecture. *Information Sciences*, 2008.

[12] F. Douglis, P. Krishnan, and B. Marsh. Thwarting the Power-Hungry Disk. In *Proceedings of the 1994 Winter USENIX Conference*, pages 293–306, 1994.

[13] E. N. Elnozahy, M. Kistler, and R. Rajamony. Energy-Efficient Server Clusters. In *Proceedings of the 2nd Workshop on Power-Aware Computing Systems*, pages 179–196, 2002.

[14] D. Essary and A. Amer. Predictive Data Grouping: Defining the Bounds of Energy and Latency Reduction through Predictive Data Grouping and Replication. *ACM Transactions on Storage (TOS)*, 4(1):1–23, May 2008.

[15] ext3. `http://en.wikipedia.org/wiki/Ext3`.

[16] FileBench, July 2008. `www.solarisinternals.com/wiki/index.php/FileBench`.

[17] A. Gulati, M. Naik, and R. Tewari. Nache: Design and Implementation of a Caching Proxy for NFSv4. In *Proceedings of the Fifth USENIX Conference on File and Storage Technologies (FAST '07)*, pages 199–214, San Jose, CA, February 2007. USENIX Association.

[18] S. Gurumurthi, J. Zhang, A. Sivasubramaniam, M. Kandemir, H. Franke, N. Vijaykrishnan, and M. J. Irwin. Interplay of Energy and Performance for Disk Arrays Running Transaction Processing Workloads. In *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 123–132, 2003.

[19] H. Huang, W. Hung, and K. Shin. FS2: Dynamic Data Replication in Free Disk Space for Improving Disk Performance and Energy Consumption. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 263–276, Brighton, UK, October 2005. ACM Press.

[20] N. Joukov and J. Sipek. GreenFS: Making Enterprise Computers Greener by Protecting Them Better. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (EuroSys 2008)*, Glasgow, Scotland, April 2008. ACM.

[21] N. Joukov, A. Traeger, R. Iyer, C. P. Wright, and E. Zadok. Operating System Profiling via Latency Analysis. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*, pages 89–102, Seattle, WA, November 2006. ACM SIGOPS.

[22] J. Katcher. PostMark: A New Filesystem Benchmark. Technical Report TR3022, Network Appliance, 1997.

[23] 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 Israeli Experimental Systems Conference (ACM SYSTOR '09)*, Haifa, Israel, May 2009. ACM.

[24] D. Li. *High Performance Energy Efficient File Storage System*. PhD thesis, Computer Science Department, University of Nebraska, Lincoln, 2006.

[25] K. Li, R. Kumpf, P. Horton, and T. Anderson. A Quantitative Analysis of Disk Drive Power Management in Portable Computers. In *Proceedings of the 1994 Winter USENIX Conference*, pages 279–291, 1994.

[26] A. Manzanares, K. Bellam, and X. Qin. A Prefetching Scheme for Energy Conservation in Parallel Disk Systems. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2008)*, pages 1–5, April 2008.

[27] R. McDougall, J. Mauro, and B. Gregg. *Solaris Performance and Tools*. Prentice Hall, New Jersey, 2007.

[28] 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.

[29] E. B. Nightingale and J. Flinn. Energy-Efficiency and Storage Flexibility in the Blue File System. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI 2004)*, pages 363–378, San Francisco, CA, December 2004. ACM SIGOPS.

[30] A. E. Papathanasiou and M. L. Scott. Increasing Disk Burstiness for Energy Efficiency. Technical Report 792, University of Rochester, 2002.

[31] 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.

[32] E. Pinheiro, R. Bianchini, E. Carrera, and T. Heath. Load Balancing and Unbalancing for Power and Performance in Cluster-Based Systems. In *International Conference on Parallel Architectures and Compilation Techniques*, Barcelona, Spain, 2001.

[33] H. Reiser. ReiserFS v.3 Whitepaper. `http://web.archive.org/web/20031015041320/http://namesys.com/`.

[34] S. Rivoire, M. A. Shah, P. Ranganathan, and C. Kozyrakis. JouleSort: A Balanced Energy-Efficiency Benchmark. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, Beijing, China, June 2007.

[35] S. Gurumurthi and A. Sivasubramaniam and M. Kandemir and H. Franke. DRPM: Dynamic Speed Control for Power Management in Server Class Disks. In *Proceedings of the 30th annual international symposium on Computer architecture*, pages 169–181, 2003.

[36] M. I. Seltzer. Transaction Support in a Log-Structured File System. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 503–510, Vienna, Austria, April 1993.

[37] SGI. XFS Filesystem Structure. `http://oss.sgi.com/projects/xfs/papers/xfs_filesystem_structure.pdf`.

[38] SPEC. SPECpower_ssj2008 v1.01. `www.spec.org/power_ssj2008/`.

[39] SPEC. SPECweb99. `www.spec.org/web99`, October 2005.

[40] SPEC. SPECsfs2008. `www.spec.org/sfs2008`, July 2008.

[41] The Standard Performance Evaluation Corporation. SPEC HPC Suite. `www.spec.org/hpc2002/`, August 2004.

[42] U.S. Environmental Protection Agency. Report to Congress on Server and Data Center Energy Efficiency. Public Law 109-431, August 2007.

[43] J. Wang, H. Zhu, and Dong Li. eRAID: Conserving Energy in Conventional Disk-Based RAID System. *IEEE Transactions on Computers*, 57(3):359–374, March 2008.

[44] D. Washburn. More Energy Is Consumed Outside Of The Data Center, 2008.

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

[46] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, 1994.

[47] C. P. Wright, N. Joukov, D. Kulkarni, Y. Miretskiy, and E. Zadok. Auto-pilot: A Platform for System Software Benchmarking. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 175–187, Anaheim, CA, April 2005. USENIX Association.

[48] OSDIR mail archive for XFS. `http://osdir.com/ml/file-systems.xfs.general/2002-06/msg00071.html`.

[49] Q. Zhu, F. M. David, C. F. Devaraj, Z. Li, Y. Zhou, and P. Cao. Reducing Energy Consumption of Disk Storage Using Power-Aware Cache Management. In *Proceedings of the 10th International Symposium on High-Performance Computer Architecture*, pages 118–129, 2004.

SRCMap: Energy Proportional Storage using Dynamic Consolidation

Akshat Verma[†] Ricardo Koller[‡] Luis Useche[‡] Raju Rangaswami[‡]
[†]*IBM Research, India*
akshatverma@in.ibm.com
[‡]*Florida International University*
{rkoll001,luis,raju}@cs.fiu.edu

Abstract

We investigate the problem of creating an energy proportional storage system through power-aware dynamic storage consolidation. Our proposal, Sample-Replicate-Consolidate Mapping (SRCMap), is a storage virtualization layer optimization that enables energy proportionality for dynamic I/O workloads by consolidating the cumulative workload on a subset of physical volumes proportional to the I/O workload intensity. Instead of migrating data across physical volumes dynamically or replicating entire volumes, both of which are prohibitively expensive, SRCMap samples a subset of blocks from each data volume that constitutes its working set and replicates these on other physical volumes. During a given consolidation interval, SRCMap activates a minimal set of physical volumes to serve the workload and spins down the remaining volumes, redirecting their workload to replicas on active volumes. We present both theoretical and experimental evidence to establish the effectiveness of SRCMap in minimizing the power consumption of enterprise storage systems.

1 Introduction

Energy Management has emerged as one of the most significant challenges faced by data center operators. The current power density of data centers is estimated to be in the range of 100 W/sq.ft. and growing at the rate of 15-20% per year [22]. Barroso and Hölzle have made the case for energy proportional computing based on the observation that servers in data centers today operate at well below peak load levels on an average [2]. A popular technique for delivering energy proportional behavior in servers is consolidation using virtualization [4, 24, 26, 27]. These techniques (a) utilize heterogeneity to select the most power-efficient servers at any given time, (b) utilize low-overhead live Virtual Machine (VM) migration to vary the number of active servers in response to workload variation, and (c) provide fine-grained control over power consumption by allowing the number of active servers to be increased or decreased one at a time.

Storage consumes roughly 10-25% of the power within computing equipment at data centers depending on the load level, consuming a greater fraction of the power when server load is lower [3]. Energy proportionality for the storage subsystem thus represents a critical gap in the energy efficiency of future data centers. In this paper, we the investigate the following fundamental question: *Can we use a storage virtualization layer to design a practical energy proportional storage system?*

Storage virtualization solutions (e.g., EMC Invista [7], HP SVSP [6], IBM SVC [12], NetApp V-Series [19]) provide a unified view of disparate storage controllers thus simplifying management [13]. Similar to server virtualization, storage virtualization provides a transparent I/O redirection layer that can be used to consolidate fragmented storage resource utilization. Similar to server workloads, storage workloads exhibit significant variation in workload intensity, motivating dynamic consolidation [16]. However, unlike the relatively inexpensive VM migration, migrating a logical volume from one device to another can be prohibitively expensive, a key factor disrupting storage consolidation solutions.

Our proposal, Sample-Replicate-Consolidate Mapping (SRCMap), is a storage virtualization layer optimization that makes storage systems energy proportional. The SRCMap architecture leverages storage virtualization to redirect the I/O workload without any changes in the hosts or storage controllers. SRCMap ties together disparate ideas from server and storage power management (namely caching, replication, transparent live migration, and write off-loading) to minimize the power drawn by storage devices in a data center. It continuously targets energy proportionality by dynamically increasing or decreasing the number of active physical volumes in a data center in response to variation in I/O workload intensity.

SRCMap is based on the following observations in production workloads detailed in §3: *(i)* the active data set in storage volumes is small, *(ii)* this active data set is stable, and *(iii)* there is substantial variation in workload intensity both within and across storage volumes.

Thus, instead of creating full replicas of data volumes, SRCMap creates partial replicas that contain the working sets of data volumes. The small replica size allows creating multiple copies on one or more target volumes or analogously allowing one target volume to host replicas of multiple source volumes. Additional space is reserved on each partial replica to offload writes [18] to volumes that are spun down.

SRCMap enables a high degree of flexibility in spinning down volumes because it activates either the primary volume or exactly one working set replica of each volume at any time. Based on the aggregate workload intensity, SRCMap changes the set of active volumes in the granularity of hours rather than minutes to address the reliability concerns related to the limited number of disk spin-up cycles. It selects active replica targets that allow spinning down the maximum number of volumes, while serving the aggregate storage workload. The virtualization layer remaps the virtual to physical volume mapping as required thereby replacing expensive data migration operations with background data synchronization operations. SRCMap is able to create close to N *power-performance levels on a storage subsystem with* N *volumes*, enabling storage energy consumption proportional to the I/O workload intensity.

In the rest of this paper, we propose design goals for energy proportional storage systems and examine existing solutions (§2), analyze storage workload characteristics (§3) that motivate design choices (§4), provide detailed system design, algorithms, and optimizations (§5 and §6), and evaluate for energy proportionality (§7). We conclude with a fairly positive view on SRCMap meeting its energy proportionality goals and some directions for future work (§8).

2 On Energy Proportional Storage

In this section, we identify the goals for a practical and effective energy proportional storage system. We also examine existing work on energy-aware storage and the extent to which they deliver on these goals.

2.1 Design Goals

1. Fine-grained energy proportionality: Energy proportional storage systems are uniquely characterized by multiple performance-power levels. True energy proportionality requires that for a system with a peak power of P_{peak} for a workload intensity ρ_{max}, the power drawn for a workload intensity ρ_i would be $P_{peak} \times \frac{\rho_i}{\rho_{max}}$.

2. Low space overhead: Replication-based strategies could achieve energy proportionality trivially by replicating each volume on all the other $N-1$ volumes. This would require N copies of each volume, representing an unacceptable space overhead. A practical energy proportional system should incur minimum space overhead; for example, 25% additional space is often available.

Design Goal	Write offloading	Caching systems	Singly Redundant	Geared RAID
Proportionality	∼	X	X	∼
Space overhead	✓	✓	X	X
Reliability	X	X	✓	✓
Adaptation	X	✓	✓	✓
Heterogeneity	∼	∼	∼	X

Table 1: **Comparison of Power Management Techniques.** ∼ *indicates the goal is partially addressed.*

3. Reliability: Disk drives are designed to survive a limited number of spin-up cycles [14]. Energy conservation based on spinning down the disk must ensure that the additional number of spin-up cycles induced during the disks' expected lifetime is significantly lesser than the manufacturer specified maximum spin-up cycles.

4. Workload shift adaptation: The popularity of data changes, even if slowly over time. Power management for storage systems that rely on caching popular data over long intervals should address any shift in popularity, while ensuring energy proportionality.

5. Heterogeneity support: A data center is typically composed of several substantially different storage systems (e.g., with variable numbers and types of drives). An ideal energy proportional storage system should account for the differences in their performance-power ratios to provide the best performance at each host level.

2.2 Examining Existing Solutions

It has been shown that the idleness in storage workload is quite low for typical server workloads [31]. We examine several classes of related work that represent approaches to increase this idleness for power minimization and evaluate the extent to which they address our design goals. We next discuss each of them and summarize their relative strengths in Table 1.

Singly redundant schemes. The central idea used by these schemes is spinning down disks with redundant data during periods of low I/O load [9, 21, 28]. RIMAC [28] uses memory-level and on-disk redundancy to reduce passive spin ups in RAID5 systems, enabling the spinning down of one out of the N disks in the array. The Diverted Accesses technique [21] generalizes this approach to find the best redundancy configuration for energy, performance, and reliability for all RAID levels. Greenan *et al.* propose generic techniques for managing power-aware erasure coded storage systems [9]. The above techniques aim to support two energy levels and do not address fine-grained energy proportionality.

Geared RAIDs. PARAID [30] is a gear-shifting mechanism (each disk spun down represents a gear shift) for a parity-based RAID. To implement $N-1$ gears in a N disk array with used storage X, PARAID requires

$O(X \log N)$ space, even if we ignore the space required for storing parity information. DiskGroup [17] is a modification of RAID-1 that enables a subset of the disks in a mirror group to be activated as necessary. Both techniques incur large space overhead. Further, they do not address heterogeneous storage systems composed of multiple volumes with varying I/O workload intensities.

Caching systems. This class of work is mostly based on caching popular data on additional storage [5, 15, 25] to spin down primary data drives. MAID [5], an archival storage system, optionally uses additional cache disks for replicating popular data to increase idle periods on the remaining disks. PDC [20] does not use additional disks but rather suggests migrating data between disks according to popularity, always keeping the most popular data on a few active disks. EXCES [25] uses a low-end flash device for caching popular data and buffering writes to increase idle periods of disk drives. Lee *et al.* [15] suggest augmenting RAID systems with an SSD for a similar purpose. A dedicated storage cache does not provide fine-grained energy proportionality; the storage system is able to save energy only when the I/O load is low and can be served from the cache. Further, these techniques do not account for the reliability impact of frequent disk spin-up operations.

Write Offloading. Write off-loading is an energy saving technique based on redirecting writes to alternate locations. The authors of write-offloading demonstrate that idle periods at a one minute granularity can be significantly increased by off-loading writes to a different volume. The reliability impact due to frequent spin-up cycles on a disk is a potential concern, which the authors acknowledge but leave as an open problem. In contrast, SRCMap increases the idle periods substantially by off-loading popular data reads in addition to the writes, and thus more comprehensively addressing this important concern. Another important question not addressed in the write off-loading work is: with multiple volumes, which active volume should be treated as a write off-loading target for each spun down volume? SRCMap addresses this question clearly with a formal process for identifying the set of active disks during each interval.

Other techniques. There are orthogonal classes of work that can either be used in conjunction with SRCMap or that address other target environments. Hibernator [31] uses DRPM [10] to create a multi-tier hierarchy of futuristic multi-speed disks. The speed for each disk is set and data migrated across tiers as the workload changes. Pergamum is an archival storage system designed to be energy-efficient with techniques for reducing inter-disk dependencies and staggering rebuild operations [23]. Gurumurthi *et al.* propose intra-disk parallelism on high capacity drives to improve disk bandwidth without increasing power consumption [11]. Finally, Ganesh *et al.* propose log-structured striped writing on a disk array to increase the predictability of active/inactive spindles [8].

Workload Volume	Size [GB]	Reads [GB] Total	Reads [GB] Uniq	Writes [GB] Total	Writes [GB] Uniq	Volume accessed
mail	500	62.00	29.24	482.10	4.18	6.27%
homes	470	5.79	2.40	148.86	4.33	1.44%
web-vm	70	3.40	1.27	11.46	0.86	2.8%

Table 2: **Summary statistics of one week I/O workload traces obtained from three different volumes.**

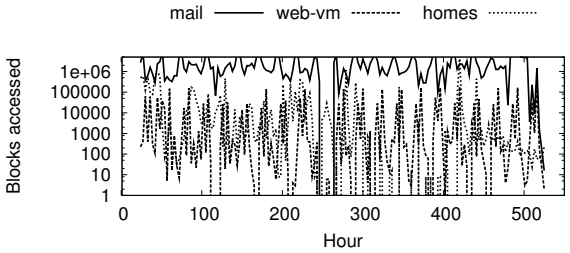


Figure 1: **Variability in I/O workload intensity.**

3 Storage Workload Characteristics

In this section, we characterize the nature of I/O access on servers using workloads from three production systems, specifically looking for properties that help us in our goal of energy proportional storage. The systems include an email server (*mail* workload), a virtual machine monitor running two web servers (*web-vm* workload), and a file server (*homes* workload). The *mail* workload serves user INBOXes for the entire Computer Science department at FIU. The *homes* workload is that of a NFS server that serves the home directories for our research group at FIU; activities represent those of a typical researcher consisting of software development, testing, and experimentation, the use of graph-plotting software, and technical document preparation. Finally, the *web-vm* workload is collected from a virtualized system that hosts two CS department web-servers, one hosting the department's online course management system and the other hosting the department's web-based email access portal.

In each system, we collected I/O traces downstream of an active page cache for a duration of three weeks. Average weekly statistics related to these workloads are summarized in Table 2. The first thing to note is that the weekly working sets (unique accesses during a week) is a small percentage of the total volume size (1.5-6.5%). This trend is consistent across all volumes and leads to our first observation.

Observation 1 *The active data set for storage volumes is typically a small fraction of total used storage.*

Dynamic consolidation utilizes variability in I/O workload intensity to increase or decrease the number of

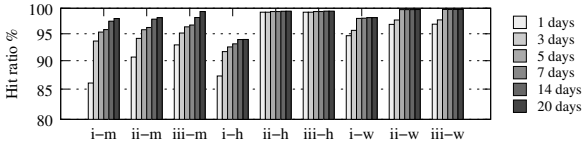


Figure 2: **Overlap in daily working sets for the mail (m), homes (h), and web-vm (w) workloads.** *(i) Reads and writes against working set , (ii) Reads against working set and (iii) Reads against working set, recently offloaded writes, and recent missed reads.*

active devices. Figure 1 depicts large variability in I/O workload intensity for each of the three workloads over time, with as much as 5-6 orders of magnitude between the lowest and highest workload intensity levels across time. This highlights the potential of energy savings if the storage systems can be made energy proportional.

Observation 2 *There is a significant variability in I/O workload intensity on storage volumes.*

Based on our first two observations, we hypothesize that there is room for powering down physical volumes that are substantially under-utilized by replicating a small active working-set on other volumes which have the spare bandwidth to serve accesses to the powered down volumes. This motivates *Sample* and *Replicate* in SRCMap. Energy conservation is possible provided the corresponding working set replicas can serve most requests to each powered down volume. This would be true if working sets are largely stable.

We investigate the stability of the volume working sets in Fig. 2 for three progressive definitions of the working set. In the first scenario, we compute the classical working set based on the last few days of access history. In the second scenario, we additionally assume that writes can be offloaded and mark all writes as hits. In the third scenario, we further expand the working set to include recent writes and past missed reads. For each scenario, we compute the working set hits and misses for the following day's workload and study the hit ratio with change in the length of history used to compute the working set. We observe that the hit ratio progressively increases both across the scenarios and as we increase the history length leading us to conclude that data usage exhibits high temporal locality and that the working set after including recent accesses is fairly stable. This leads to our third observation (also observed earlier by Leung *et al.* [16]).

Observation 3 *Data usage is highly skewed with more than* 99% *of the working set consisting of some 'really popular' data and 'recently accessed' data.*

The first three observations are the pillars behind the *Sample*, *Replicate* and *Consolidate* approach whereby we sample each volume for its working set, replicate

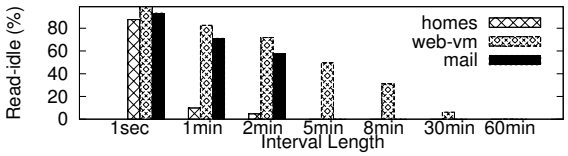


Figure 3: **Distribution of read-idle times.**

these working sets on other volumes, and consolidate I/O workloads on proportionately fewer volumes during periods of low load. Before designing a new system based on the above observations, we study the suitability of a simpler *write-offloading* technique for building energy proportional storage systems. Write off-loading is based on the observation that I/O workloads are write dominated and simply off-loading writes to a different volume can cause volumes to be idle for a substantial fraction (79% for workloads in the original study) of time [18]. While write off-loading increases the fraction of idle time of volumes, the distribution of idle time durations due to write off-loading raises an orthogonal, but important, concern. If these idle time durations are short, saving power requires frequent spinning down/up of the volumes which degrades reliability of the disk drives.

Figure 3 depicts the read-idle time distributions of the three workloads. It is interesting to note that idle time durations for the *homes* and *mail* workloads are all less than or equal to 2 minutes, and for the *web-vm* the majority are less than or equal to 5 minutes are all are less than 30 minutes.

Observation 4 *The read-idle time distribution (periods of writes alone with no intervening read operations) of I/O workloads is dominated by small durations, typically less than five minutes.*

This observation implies that exploiting all read-idleness for saving power will necessitate spinning up the disk at least 720 times a day in the case of *homes* and *mail* and at least 48 times in the case of *web-vm*. This can be a significant hurdle to reliability of the disk drives which typically have limited spin-up cycles [14]. It is therefore important to develop new techniques that can substantially increase average read-idle time durations.

4 Background and Rationale

Storage virtualization managers simplify storage management by enabling a uniform view of disparate storage resources in a data center. They export a storage controller interface allowing users to create logical volumes or virtual disks (*vdisks*) and mount these on hosts. The physical volumes managed by the physical storage controllers are available to the virtualization manager as managed disks (*mdisks*) entirely transparently to the

hosts which only view the logical *vdisk* volumes. A useful property of the virtualization layer is the complete flexibility in allocation of $mdisk$ extents to $vdisk$s.

Applying server consolidation principles to storage consolidation using virtualization would activate only the most energy-efficient $mdisk$s required to serve the aggregate workload during any period T. Data from the other $mdisk$s chosen to be spun down would first need to be migrated to active $mdisk$s to effect the change. While data migration is an expensive operation, the ease with which virtual-to-physical mappings can be reconfigured provides an alternative approach. A naïve strategy following this approach could replicate data for each $vdisk$ on all the $mdisk$s and adapt to workload variations by dynamically changing the virtual-to-physical mappings to use only the selected $mdisk$s during T. Unfortunately, this strategy requires N times additional space for a N $vdisk$ storage system, an unacceptable space overhead.

SRCMap intelligently uses the storage virtualization layer as an I/O indirection mechanism to deliver a practically feasible, energy proportional solution. Since it operates at the storage virtualization manager, it does not alter the basic redundancy-based reliability properties of the underlying physical volumes which is determined by the respective physical volume (e.g., RAID) controllers. To maintain the redundancy level, SRCMap ensures that a volume is replicated on target volumes at the same RAID level. While we detail SRCMap's design and algorithms in subsequent sections (§ 5 and § 6), here we list the rationale behind SRCMap's design decisions. These design decisions together help to satisfy the design goals for an ideal energy proportional storage system.

I. Multiple replica targets. Fine-grained energy proportionality requires the flexibility to increase or decrease the number of active physical volumes one at a time. Techniques that activate a fixed secondary device for each data volume during periods of low activity cannot provide the flexibility necessary to deactivate an arbitrary fraction of the physical volumes. In SRCMap, we achieve this fine-grained control by creating a primary $mdisk$ for each $vdisk$ and replicating only the working set of each $vdisk$ on multiple secondary $mdisk$s. This ensures that *(a)* every volume can be offloaded to one of multiple targets and *(b)* each target can serve the I/O workload for multiple $vdisk$s. During peak load, each $vdisk$ maps to its primary $mdisk$ and all $mdisk$s are active. However, during periods of low activity, SRCMap selects a proportionately small subset of $mdisk$s that can support the aggregate I/O workload for all $vdisk$s.

II. Sampling. Creating multiple full replicas of $vdisk$s is impractical. Drawing from *Observation 1* (§ 3), SRCMap substantially reduces the space overhead of maintaining multiple replicas by sampling only the working set for each $vdisk$ and replicating it. Since the working set is typically small , the space overhead is low.

III. Ordered replica placement. While sampling helps to reduce replica sizes substantially, creating multiple replicas for each sample still induces space overhead. In SRCMap, we observe that all replicas are not created equal; for instance, it is more beneficial to replicate a lightly loaded volume than a heavily loaded one which is likely to be active anyway. Similarly, a large working set has greater space overhead; SRCMap chooses to create fewer replicas aiming to keep it active, if possible. As we shall formally demonstrate, carefully ordering the replica placement helps to minimize the number of active disks for fine-grained energy proportionality.

IV. Dynamic source-to-target mapping and dual data synchronization. From *Observation 2* (§ 3), we know that workloads can vary substantially over a period of time. Hence, it is not possible to pre-determine which volumes need to be active. Target replica selection for any volume being powered down therefore needs to be a dynamic decision and also needs to take into account that some volumes have more replicas (or target choices) than others. We use two distinct mechanisms for updating the replica working sets. The active replica lies in the data path and is immediately synchronized in the case of a *read miss*. This ensures that the active replica continuously *adapts* with change in *workload popularity*. The secondary replicas, on the other hand, use a lazy, incremental data synchronization in the background between the primary replica and any secondary replicas present on active $mdisk$s. This ensures that switching between replicas requires minimal data copying and can be performed fairly quickly.

V. Coarse-grained power cycling. In contrast to most existing solutions that rely on fine-grained disk power-mode switching, SRCMap implements coarse-grained consolidation intervals (of the order of hours), during each of which the set of active $mdisk$s chosen by SRCMap does not change. This ensures normal disk lifetimes are realized by adhering to the disk power cycle specification contained within manufacturer data sheets.

5 Design Overview

SRCMap is built in a modular fashion to directly interface with storage virtualization managers or be integrated into one as shown in Figure 4. The overall architecture supports the following distinct flows of control:

(i) the *replica generation flow* (Flow A) identifies the working set for each $vdisk$ and replicates it on multiple $mdisk$s. This flow is orchestrated by the *Replica Placement Controller* and is triggered once when SRCMap

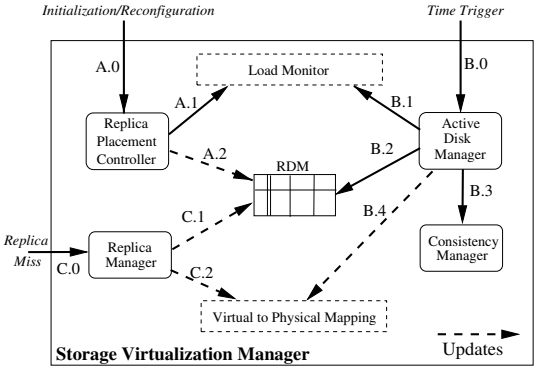


Figure 4: **SRCMap integrated into a Storage Virtualization Manager.** *Arrows depict control flow. Dashed/solid boxes denote existing/new components.*

is initialized and whenever a configuration change (e.g., addition of a new workload or new disks) takes place. Once a trigger is generated, the *Replica Placement Controller* obtains a historical workload trace from the *Load Monitor* and computes the working set and the long-term workload intensity for each volume ($vdisk$). The working set is then replicated on one or more physical volumes ($mdisks$). The blocks that constitute the working set for the $vdisk$ and the target physical volumes where these are replicated are managed using a common data structure called the *Replica Disk Map (RDM)*.

(ii) the *active disk identification flow* (Flow B) identifies, for a period T, the active $mdisk$s and activated replicas for each inactive $mdisk$. The flow is triggered at the beginning of the consolidation interval T (e.g., every 2 hours) and orchestrated by the *Active Disk Manager*. In this flow, the *Active Disk Manager* queries the *Load Monitor* for expected workload intensity of each $vdisk$ in the period T. It then uses the workload information along with the placement of working set replicas on target $mdisks$ to compute the set of active primary $mdisk$s and a active secondary replica $mdisk$ for each inactive primary $mdisk$. It then directs the *Consistency Manager* to ensure that the data on any selected active primary or active secondary replica is current. Once consistency checks are made, it updates the *Virtual to Physical Mapping* to redirect the workload to the appropriate $mdisk$.

(iii) the *I/O redirection flow* (Flow C) is an extension of the I/O processing in the *storage virtualization manager* and utilizes the built-in virtual-to-physical re-mapping support to direct requests to primaries or active replicas. Further, this flow ensures that the working-set of each $vdisk$ is kept up-to-date. To ensure this, whenever a request to a block not available in the active replica is made, a *Replica Miss* event is generated. On a *Replica Miss*, the *Replica Manager* spin-ups the primary $mdisk$ to fetch the required block. Further, it adds this new block to the working set of the $vdisk$ in the RDM. We next describe the key components of SRCMap.

5.1 Load Monitor

The *Load Monitor* resides in the storage virtualization manager and records access to data on any of the $vdisk$s exported by the virtualization layer. It provides two interfaces for use by SRCMap – long-term workload data interface invoked by the *Replica Placement Controller* and predicted short-term workload data interface invoked by the *Active Disk Manager*.

5.2 Replica Placement Controller

The *Replica Placement Controller* orchestrates the process of *Sampling* (identifying working sets for each $vdisk$) and *Replicating* on one or more target $mdisk$s. We use a conservative definition of working set that includes all the blocks that were accessed during a fixed duration, configured as the minimum duration beyond which the hit ratio on the working set saturates. Consequently, we use 20 days for $mail$, 14 days for $homes$ and 5 days for $web\text{-}vm$ workload (Fig. 2). The blocks that capture the working set for each $vdisk$ and the $mdisks$ where it is replicated are stored in the RDM. The details of the parameters and methodology used within *Replica Placement* are described in Section 6.1.

5.3 Active Disk Manager

The *Active Disk Manager* orchestrates the *Consolidate* step in SRCMap. The module takes as input the workload intensity for each $vdisk$ and identifies if the primary $mdisk$ can be spun down by redirecting the workload to one of the secondary $mdisk$s hosting its replica. Once the target set of active $mdisk$s and replicas are identified, the *Active Disk Manager* synchronizes the identified active primaries or active secondary replicas and updates the virtual-to-physical mapping of the storage virtualization manager, so that I/O requests to a $vdisk$ could be redirected accordingly. The Active Disk Manager uses a *Consistency Manager* for the synchronization operation. Details of the algorithm used by *Active Disk Manager* for selecting active $mdisks$ are described in Section 6.2.

5.4 Consistency Manager

The *Consistency Manager* ensures that the primary $mdisk$ and the replicas are consistent. Before an $mdisk$ is spun down and a new replica activated, the new active replica is made consistent with the previous one. In order to ensure that the overhead during the re-synchronization is minimal, an incremental point-in-time (PIT) relationship (e.g., Flash-copy in IBM SVC [12]) is maintained between the active data (either the primary $mdisk$ or one of the active replicas) and all other copies of the same data. A *go-to-sync* operation is performed periodically between the active data and all its copies on active $mdisk$s. This ensures that when an $mdisk$ is spun up or down, the amount of data to be synchronized is small.

5.5 Replica Manager

The *Replica Manager* ensures that the replica data set for a $vdisk$ is able to mimic the working set of the $vdisk$ over time. If a data block unavailable at the active replica of the $vdisk$ is read causing a *replica miss*, the Replica Manager copies the block to the replica space assigned to the active replica and adds the block to the *Replica Metadata* accordingly. Finally, the *Replica Manager* uses a Least Recently Used (LRU) policy to evict an older block in case the replica space assigned to a replica is filled up. If the active data set changes drastically, there may be a large number of *replica misses*. All these replica misses can be handled by a single spin-up of the primary $mdisk$. Once all the data in the new working set is touched, the primary $mdisk$ can be spun-down as the active replica is now up-to-date. The continuous updating of the *Replica Metadata* enables SRCMap to meet the goal of *Workload shift adaptation*, without re-running the expensive *replica generation flow*. The *replica generation flow* needs to re-run only when a disruptive change occurs such as addition of a new workload or a new volume or new disks to a volume.

6 Algorithms and Optimizations

In this section, we present details about the algorithms employed by SRCMap. We first present the long-term replica placement methodology and subsequently, the short-term active disk identification method.

6.1 Replica Placement Algorithm

The *Replica Placement Controller* creates one or more replicas of the working set of each $vdisk$ on the available replica space on the target $mdisk$s. We use the insight that all replicas are not created equal and have distinct associated costs and benefits. The space cost of creating the replica is lower if the $vdisk$ has a smaller working set. Similarly, the benefit of creating a replica is higher if the $vdisk$ (i) has a stable working set (lower misses if the primary $mdisk$ is switched off), (ii) has a small average load making it easy to find spare bandwidth for it on any target $mdisk$, and (iii) is hosted on a less power-efficient primary $mdisk$. Hence, the goal of both *Replica Placement* and *Active Disk Identification* is to ensure that we create more replicas for $vdisk$s that have a favorable cost-benefit ratio. The goal of the replica placement is to ensure that if the *Active Disk Manager* decides to spin down the primary $mdisk$ of a $vdisk$, it should be able to find at least one active target $mdisk$ that hosts its replica, captured in the following *Ordering Property*.

Definition 1 Ordering Property*: For any two $vdisks$ V_i and V_j, if V_i is more likely to require a replica target than V_j at any time t during* Active Disk Identification*, then V_i is more likely than V_j to find a replica target amongst active $mdisks$ at time t.*

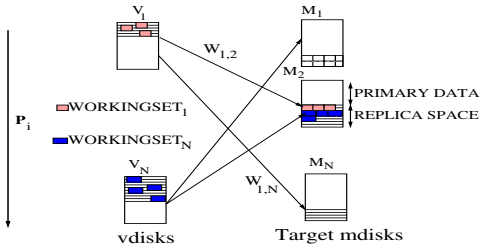


Figure 5: **Replica Placement Model**

The *replica placement algorithm* consists of (i) creating an initial ordering of $vdisk$s in terms of cost-benefit tradeoff (ii) a bipartite graph creation that reflects this ordering (iii) iteratively creating one source-target mapping respecting the current order and (iv) re-calibration of edge weights to ensure the *Ordering Property* holds for the next iteration of source-target mapping.

6.1.1 Initial $vdisk$ ordering

The Initial $vdisk$ ordering creates a sorted order amongst $vdisk$s based on their cost-benefit tradeoff. For each $vdisk$ V_i, we compute the probability P_i that its primary $mdisk$ M_i would be spun down as

$$P_i = \frac{w_1 WS_{min}}{WS_i} + \frac{w_2 PPR_{min}}{PPR_i} + \frac{w_3 \rho_{min}}{\rho_i} + \frac{w_f m_{min}}{m_i} \quad (1)$$

where the w_k are tunable weights, WS_i is the size of the working set of V_i, PPR_i is the performance-power ratio (ratio between the peak IO bandwidth and peak power) for the primary mdisk M_i of V_i, ρ_i is the average long-term I/O workload intensity (measured in IOPS) for V_i, and m_i is the number of read misses in the working set of V_i, normalized by the number of spindles used by its primary mdisk M_i. The corresponding min subscript terms represent the minimum values across all the $vdisk$s and provide normalization. The probability formulation is based on the dual rationale that it is relatively easier to find a target $mdisk$ for a smaller workload and switching off relatively more power-hungry disks saves more power. Further, we assign a higher probability for spinning down $mdisk$s that host more stable working sets by accounting for the number of times a read request cannot be served from the replicated working set, thereby necessitating the spinning up of the primary $mdisk$.

6.1.2 Bipartite graph creation

Replica Placement creates a bipartite graph $G(V \rightarrow M)$ with each $vdisk$ as a source node V_i, its primary $mdisk$ as a target node M_i, and the edge weights $e(V_i, M_j)$ representing the cost-benefit trade-off of placing a replica of V_i on M_j (Fig. 5). The nodes in the bipartite graph are sorted using P_i (disks with larger P_i are at the top). We initialize the edge weights $w_{i,j} = P_i$ for each edge $e(V_i, M_j)$ (source-target pair). Initially, there are no

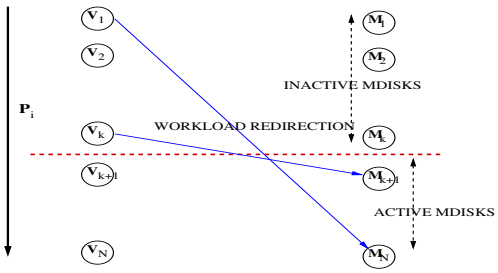


Figure 6: **Active Disk Identification**

replica assignments made to any target $mdisk$. The replica placement algorithm iterates through the following two steps, until all the available replica space on the target $mdisk$s have been assigned to source $vdisk$ replicas. In each iteration, exactly one target $mdisk$'s replica space is assigned.

6.1.3 Source-Target mapping

The goal of the replica placement method is to achieve a source target mapping that achieves the *Ordering property*. To achieve this goal, the algorithm takes the top-most target $mdisk$ M_i whose replica space is not yet assigned and selects the set of highest weight incident edges such that the combined replica size of the source nodes in this set fills up the replica space available in M_i (e.g, the working sets of V_1 and V_N are replicated in the replica space of M_2 in Fig. 5). When the replica space on a target $mdisk$ is filled up, we mark the target $mdisk$ as assigned. One may observe that this procedure always gives preference to source nodes with a larger P_i. Once an $mdisk$ finds a replica, the likelihood of it requiring another replica decreases and we factor this using a re-calibration of edge weights, which is detailed next.

6.1.4 Re-calibration of edge weights

We observe that the initial assignments of weights ensure the *Ordering property*. However, once the working set of a $vdisk$ V_i has been replicated on a set of target $mdisk$s $\mathbf{T_i} = M_1, \ldots, M_{least}$ (M_{least} is the $mdisk$ with the least P_i in $\mathbf{T_i}$) s.t. $P_i > P_{least}$, the probability that V_i would require a new target $mdisk$ during *Active Disk Identification* is the probability that both M_i and M_{least} would be spun down. Hence, to preserve the *Ordering property*, we re-calibrate the edge weights of all outgoing edges of any primary $mdisk$s S_i assigned to target mdisks T_j as

$$\forall k \quad w_{i,k} = P_j P_i \tag{2}$$

Once the weights are recomputed, we iterate from the Source-Target mapping step until all the replicas have been assigned to target $mdisk$s. One may observe that the re-calibration succeeds in achieving the *Ordering* property because we start assigning the replica space for the top-most target $mdisk$s first. This allows us to increase the weights of source nodes monotonically as we place more replicas of its working set. We formally prove the following result in the appendix.

S = set of disks to be spun down
A = set of disks to be active
Sort S by reverse of P_i
Sort A by P_i
For each $D_i \in S$
  For each $D_j \in A$
    If D_j hosts a replica R_i of D_i AND
    D_j has spare bandwidth for R_i
      $Candidate(D_i) = D_j,\ break$
  End-For
  If $Candidate(D_i)$==$null$ return $Failure$
End-for
$\forall i,\ D_i \in S$ return $Candidate(D_i)$

Figure 7: **Active Replica Identification algorithm**

Theorem 1 *The* Replica Placement Algorithm *ensures ordering property.*

6.2 Active Disk Identification

We now describe the methodology employed to identify the set of active $mdisk$s and replicas at any given time. For ease of exposition, we define the probability P_i of a primary $mdisk$ M_i equal to the probability P_i of its $vdisk$ V_i. Active disk identification consists of:

I: Active $mdisk$ Selection: We first estimate the expected aggregate workload to the storage subsystem in the next interval. We use the workload to a $vdisk$ in the previous interval as the predicted workload in the next interval for the $vdisk$. The aggregate workload is then estimated as sum of the predicted workloads for all $vdisk$s in the storage system. This aggregate workload is then used to identify the minimum subset of $mdisk$s (ordered by reverse of P_i) such that the aggregate bandwidth of these $mdisk$s exceeds the expected aggregate load.
II: Active Replica Identification: This step elaborated shortly identifies one (of the many possible) replicas on an active $mdisk$ for each inactive $mdisk$ to serve the workload redirected from the inactive $mdisk$.
III: Iterate: If the Active Replica Identification step succeeds in finding an active replica for all the inactive $mdisk$s, the algorithm terminates. Else, the number of active $mdisk$s are increased by 1 and the algorithm repeats the Active Replica Identification step.

One may note that since the number of active disks are based on the maximum predicted load in a consolidation interval, a sudden increase in load may lead to an increase in response times. If performance degradation beyond user-defined acceptable levels persists beyond a user-defined interval (e.g, 5 mins), the *Active Disk Identification* is repeated for the new load.

6.2.1 Active Replica Identification

Fig. 6 depicts the high-level goal of Active Replica Identification, which is to have the primary $mdisk$s for

$vdisk$s with larger P_i spun down, and their workload directed to few $mdisk$s with smaller P_i. To do so, it must identify an active replica for each inactive primary $mdisk$, on one of the active $mdisk$s. The algorithm uses two insights: (i) The *Replica Placement* process creates more replicas for $vdisk$s with a higher probability of being spun down (P_i) and (ii) primary $mdisk$s with larger P_i are likely to be spun down for a longer time.

To utilize the first insight, we first allow primary $mdisk$s with small P_i, which are marked as inactive, to find an active replica, as they have fewer choices available. To utilize the second insight, we force inactive primary $mdisk$s with large P_i to use a replica on active $mdisk$s with small P_i. For example in Fig. 6, $vdisk$ V_k has the first choice of finding an active $mdisk$ that hosts its replica and in this case, it is able to select the first active $mdisk$ M_{k+1}. As a result, inactive $mdisk$s with larger P_i are mapped to active $mdisk$s with the smaller P_i (e.g, V_1 is mapped to M_N). Since an $mdisk$ with the smallest P_i is likely to remain active most of the time, this ensures that there is little to no need to 'switch active replicas' frequently for the inactive disks. The details of this methodology are described in Fig. 7.

6.3 Key Optimizations to Basic SRCMap

We augment the basic SRCMap algorithm to increase its practical usability and effectiveness as follows.

6.3.1 Sub-volume creation

SRCMap redirects the workload for any primary $mdisk$ that is spun down to exactly one target $mdisk$. Hence, a target $mdisk$ M_j for a primary $mdisk$ M_i needs to support the combined load of the vdisks V_i and V_j in order to be selected. With this requirement, the SRCMap consolidation process may incur a fragmentation of the available I/O bandwidth across all volumes. To elaborate, consider an example scenario with 10 identical $mdisk$s, each with capacity C and input load of $C/2 + \delta$. Note that even though this load can be served using $10/2 + 1$ $mdisk$s, there is no single $mdisk$ can support the input load of 2 $vdisk$s. To avoid such a scenario, SRCMap sub-divides each $mdisk$ into N_{SV} sub-volumes and identifies the working set for each sub-volume separately. The sub-replicas (working sets of a sub-volume) are then placed independently of each other on target $mdisk$s. With this optimization, SRCMap is able to subdivide the least amount of load that can be migrated, thereby dealing with the fragmentation problem in a straightforward manner.

This optimization requires a complementary modification to the*Replica Placement* algorithm. The Source-Target mapping step is modified to ensure that sub-replicas belonging to the same source $vdisk$ are not co-located on a target $mdisk$.

6.3.2 Scratch Space for Writes and Missed Reads

SRCMap incorporates the basic write off-loading mechanism as proposed by Narayanan *et al.* [18]. The current implementation of SRCMap uses an additional allocation of write scratch space with each sub-replica to absorb new writes to the corresponding portion of the data volume. A future optimization is to use a single write scratch space within each target $mdisk$ rather than one per sub-replica within the target $mdisk$ so that the overhead for absorbing writes can be minimized.

A key difference from write off-loading, however, is that on a *read miss* for a spun down volume, SRCMap additionally offloads the data read to dynamically learn the working-set. This helps SRCMap achieve the goal of *Workload Shift Adaptation* with change in working set. While write off-loading uses the inter read-miss durations exclusively for spin down operations, SRCMap targets capturing entire working-sets including both reads and writes in replica locations to prolong read-miss durations to the order of hours and thus places more importance on learning changes in the working-set.

7 Evaluation

In this section, we evaluate SRCMap using a prototype implementation of SRCMap-based storage virtualization manager and an energy simulator seeded by the prototype. We investigate the following questions:

1. What degree of proportionality in energy consumption and I/O load can be achieved using SRCMap?
2. How does SRCMap impact reliability?
3. What is the impact of storage consolidation on the I/O performance?
4. How sensitive are the energy savings to the amount of over-provisioned space?
5. What is the overhead associated with implementing an SRCMap indirection optimization?

Workload The workloads used consist of I/O requests to eight independent data volumes, each mapped to an independent disk drive. In practice, volumes will likely comprise of more than one disk, but resource restrictions did not allow us to create a more expansive testbed. We argue that *relative* energy consumption results still hold despite this approximation. These volumes support a mix of production web-servers from the FIU CS department data center, end-user $homes$ data, and our lab's Subversion (SVN) and Wiki servers as detailed in Table 3.

Workload I/O statistics were obtained by running *blktrace* [1] on each volume. Observe that there is a wide variance in their load intensity values, creating opportunities for consolidation across volumes.

Storage Testbed For experimental evaluation, we set up a single machine (Intel Pentium 4 HT 3GHz, 1GB mem-

Volume	ID	Disk Model	Size [GB]	Avg IOPS	Max IOPS
home-1	D0	WD5000AAKB	270	8.17	23
online	D1	WD360GD	7.8	22.62	82
webmail	D2	WD360GD	7.8	25.35	90
webresrc	D3	WD360GD	10	7.99	59
webusers	D4	WD360GD	10	18.75	37
svn-wiki	D5	WD360GD	20	1.12	4
home-2	D6	WD2500AAKS	170	0.86	4
home-3	D7	WD2500AAKS	170	1.37	12

Table 3: **Workload and storage system details.**

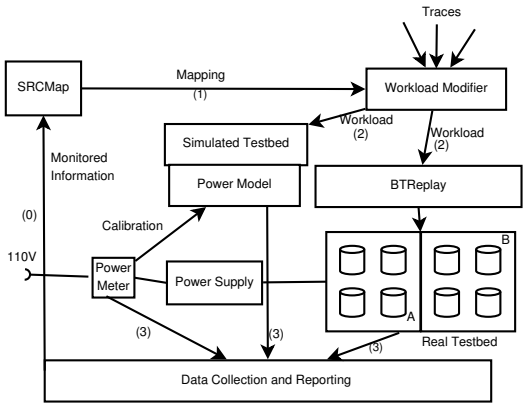


Figure 8: **Logical view of experimental setup**

ory) connected to 8 disks via two SATA-II controllers A and B. The cumulative (merged workload) trace is played back using *btreplay* [1] with each volume's trace played back to the corresponding disk. All the disks share one power supply P that is dedicated only for the experimental drives; the machine connects to another power supply. The power supply P is connected to a *Watts up? PRO* power meter [29] which allows us to measure power consumption at a one second granularity with a resolution of 0.1W. An overhead of 6.4W is introduced by the power supply itself which we deduct from all our power measurements.

Experimental Setup We describe the experimental setup used in our evaluation study in Fig. 8. We implemented an SRCMap module with its algorithms for replica placement and active disk identification during any consolidation interval. An overall experimental run consists of using the monitored data to (1) identify the consolidation candidates for each interval and create the virtual-to-physical mapping (2) modify the original traces to reflect the mapping and replaying it, and (3) power and response time reporting. At each consolidation event, the *Workload Modifier* generates the necessary additional I/O to synchronize data across the subvolumes affected due to active replica changes.

We evaluate SRCMap using two different sets of experiments: *(i)* prototype runs and *(ii)* simulated runs. The prototype runs evaluate SRCMap against a real storage system and enable realistic measurements of power consumption and impact to I/O performance via the reporting module. In a prototype run, the modified I/O workload is replayed on the actual testbed using *btreplay* [1].

Volume ID	L(0) [IOPS]	L(1) [IOPS]	L(2) [IOPS]	L(3) [IOPS]	L(4) [IOPS]
D0	33	57	74	96	125
D1-D5	52	89	116	150	196
D6, D7	38	66	86	112	145

(a)

0	1	2	3	4	5	6	7	8
19.8	27.2	32.7	39.1	44.3	49.3	55.7	59.7	66.1

(b)

Table 4: **Experimental settings: (a) Estimated disk IOPS capacity levels. (b) Storage system power consumption in Watts as the number of disks in active mode are varied from 0 to 8.** *All disks consumed approximately the same power when active. The disks not in active mode consume standby power which was found to be the same across all disks.*

The simulator runs operate similarly on a simulated testbed, wherein a power model instantiated with power measurements from the testbed is used for reporting the power numbers. The advantage with the simulator is the ability to carry out longer duration experiments in simulated time as opposed to real-time allowing us to explore the parameter space efficiently. Further, one may use it to simulate various types of storage testbeds to study the performance under various load conditions. In particular, we use the simulator runs to evaluate energy-proportionality by simulating the testbed with different values of disk IOPS capacity estimates. We also simulate alternate power management techniques (e.g., caching, replication) for a comparative evaluation.

All experiments with the prototype and the simulator were performed with the following configuration parameters. The consolidation interval was chosen to be 2 hours for all experiments to restrict the worst-case spin-up cycles for the disk drives to an acceptable value. Two minute disk timeouts were used for inactive disks; active disks within a consolidation interval remain continuously active. Working sets and replicas were created based on a three week workload history and we report results for a subsequent 24 hour duration for brevity. The consolidation is based on an estimate of the disk IOPS capacity, which varies for each volume. We computed an estimate of the disk IOPS using a synthetic random I/O workload for each volume separately (Level $L1$). We use 5 IOPS estimation levels (L0 through L4) to (a) simulate storage testbeds at different load factors and (b) study the sensitivity of SRCMap with the volume IOPS estimation. The per volume sustainable IOPS at each of these load levels is provided in Table 4(a). The power consumption of the storage system with varying number of disks in active mode is presented in Table 4(b).

7.1 Prototype Results

For the prototype evaluation, we took the most dynamic 8-hour period (4 consolidation intervals) from the

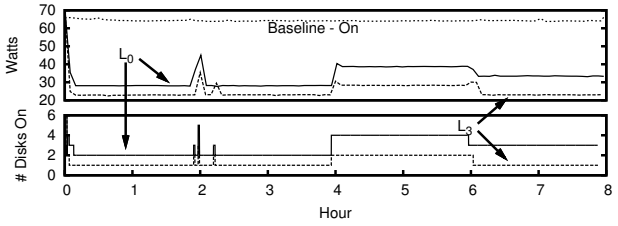


Figure 9: **Power and active disks time-line.**

24 hours and played back I/O traces for the 8 workloads described earlier in real-time. We report actual power consumption and the I/O response time (which includes queuing and service time) distribution for SRCMap when compared to a baseline configuration where all disks are continuously active. Power consumption was measured every second and disk active/standby state information was polled every 5 seconds. We used 2 different IOPS levels; $L0$ when a very conservative (low) estimate of the disk IOPS capacity is made and $L3$ when a reasonably aggressive (high) estimate is made.

We study the power savings due to SRCMap in Figure 9. Even using a conservative estimate of disk IOPS, we are able to spin down approximately 4.33 disks on an average, leading to an average savings of $23.5W$ (35.5%). Using an aggressive estimate of disk IOPS, SRCMap is able to spin down 7 disks saving $38.9W$ (59%) for all periods other than the 4hr-6hr period. In the 4-6 hr period, it uses 2 disks leading to a power savings of $33.4W$ (50%). The spikes in the power consumption relate to planned and unplanned (due to read misses) volume activations, which are few in number. It is important to note that substantial power is used in maintaining standby states ($19.8W$) and within the dynamic range, the power savings due to SRCMap are even higher.

We next investigate any performance penalty incurred due to consolidation. Fig. 10 (upper) depicts the cumulative probability density function (CDF) of response times for three different configurations: *Baseline - On* – no consolidation and all disks always active, SRCMap using L_0, and L_3. The accuracy of the CDFs for L_0 and L_3 suffer from a reporting artifact that the CDFs include the latencies for the synchronization I/Os themselves which we were not able to filter out. We throttle the synchronization I/Os to one every 10ms to reduce their interference with foreground operations.

First, we observed that less than 0.003% of the requests incurred a spin-up hit due to read misses resulting in latencies of greater than 4 seconds in both the L_0 and L_3 configurations (not shown). This implies that the working-set dynamically updated with missed reads and offloaded writes is a fairly at capturing the active data for these workloads. Second, we observe that for response times greater than 1ms, *Baseline - On* demonstrates better performance than L_0 and L_3 (upper plot). For both L_0 and L_3, less than 8% of requests incur latencies greater than 10ms, less than 2% of requests incur latencies greater than 100ms. L_0, having more disks at its disposal, shows slightly better response times than L_3. For response times lower than 1ms a reverse trend is observed wherein the SRCMap configurations do better than *Baseline - On* . We conjectured that this is due to the influence of the low latency writes during synchronization operations.

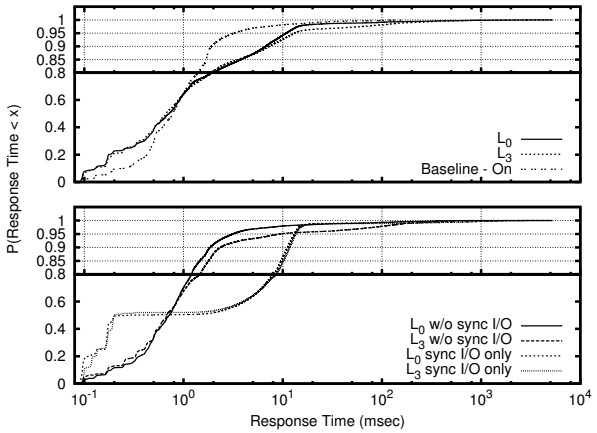


Figure 10: **Impact of consolidation on response time.**

To further delineate the influence of synchronization I/Os, we performed two additional runs. In the first run, we disable all synchronization I/Os and in the second, we disable all foreground I/Os (lower plot). The CDFs of only the synchronization operations, which show a bimodal distribution with 50% low-latency writes absorbed by the disk buffer and 50% reads with latencies greater than 1.5ms, indicate that synchronization reads are contributing towards the increased latencies in L_0 and L_3 for the upper plot. The CDF without synchronization ('w/o synch') is much closer to *Baseline - On* with a decrease of approximately 10% in the number of request with latencies greater than 1ms. Intelligent scheduling of synchronization I/Os is an important area of future work to further reduce the impact on foreground I/O operations.

7.2 Simulator Results

We conducted several experiments with simulated testbeds hosting disks of capacities $L0$ to $L4$. For brevity, we report our observations for disk capacity levels $L0$ and $L3$, expanding to other levels only when required.

7.2.1 Comparative Evaluation

We first demonstrate the basic energy proportionality achieved by SRCMap in its most conservative configuration (L0) and three alternate solutions, *Caching-1*, *Caching-2*, and *Replication*. *Caching-1* is a scheme that uses 1 additional physical volume as a cache. If the aggregate load observed is less than the IOPS capacity of

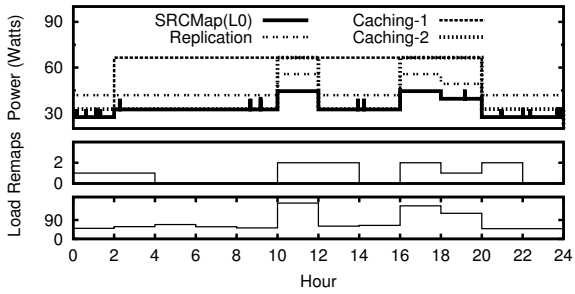


Figure 11: **Power consumption, remap operations, and aggregate load across time for a single day.**

the cache volume, the workload is redirected to the cache volume. If the load is higher, the original physical volumes are used. *Caching-2* uses 2 cache volumes in a similar manner. *Replication* identifies pairs of physical volumes with similar bandwidths and creates replica pairs, where all the data on one volume is replicated on the other. If the aggregate load to a pair is less than the IOPS capacity of one volume, only one in the pair is kept active, else both volumes are kept active.

Figure 11 evaluates power consumption of all four solutions by simulating the power consumed as volumes are spun up/down over 12 2-hour consolidation intervals. It also presents the average load (measured in IOPS) within each consolidation interval. In the case of SRCMap, read misses are indicated by instantaneous power spikes which require activating an additional disk drive. To avoid clutter, we do not show the spikes due to read misses for the Cache-1/2 configurations. We observe that each of solutions demonstrate varying degrees of energy proportionality across the intervals. SRCMap (L0) uniformly consumes the least amount of power across all intervals and its power consumption is proportional to load. Replication also demonstrates good energy proportionality but at a higher power consumption on an average. The caching configurations are the least energy proportional with only two effective energy levels to work with.

We also observe that SRCMap remaps (i.e., changes the active replica for) a minimal number of volumes – either 0, 1, or 2 during each consolidation interval. In fact, we found that for all durations the number of volumes being remapped equaled the change in the number of active physical volumes. indicating that the number of synchronization operations are kept to the minimum. Finally, in our system with eight volumes, Caching-1, Caching-2, and Replication use 12.5%, 25% and 100% additional space respectively, while as we shall show later, SRCMap is able to deliver almost all its energy savings with just 10% additional space.

Next, we investigate how SRCMap modifies per-volume activity and power consumption with an aggressive configuration L3, a configuration that demonstrated

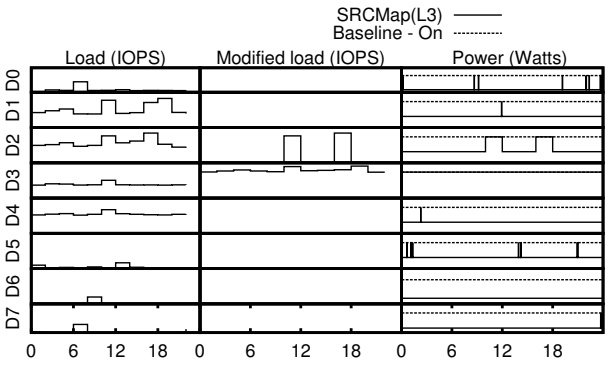


Figure 12: **Load and power consumption for each disk.** *Y ranges for all loads is $[1 : 130]$ IOPS in logarithmic scale. Y ranges for power is $[0 : 19]$ W.*

interesting consolidation dynamics over the 12 2-hour consolidation intervals. Each row in Figure 12 is specific to one of the eight volumes $D0$ through $D7$. The left and center columns show the original and SRCMap-modified load (IOPS) for each volume. The modified load were consolidated on disks $D2$ and $D3$ by SRCMap. Note that disks $D6$ and $D7$ are continuously in standby mode, $D3$ is continuously in active mode throughout the 24 hour duration while the remaining disks switched states more than once. Of these, $D0$, $D1$ and $D5$ were maintained in standby mode by SRCMap, but were spun up one or more times due to read misses to their replica volumes, while $D2$ was made active by SRCMap for two of the consolidation intervals only.

We note that the number of spin-up cycles did not exceed 6 for any physical volume during the 24 hour period, thus not sacrificing reliability. Due to the reliability-aware design of SRCMap, volumes marked as active consume power even when there is idleness over shorter, sub-interval durations. For the right column, power consumption for each disk in either active mode or spun down is shown with spikes representing spin-ups due to read misses in the volume's active replica. Further, even if the working set changes drastically during an interval, it only leads to a single spin up that services a large number of misses. For example, $D1$ served approximately $5*10^4$ misses in the single spin-up it had to incur (Figure omitted due to lack of space). We also note that summing up power consumption of individual volumes cannot be used to compute total power as per Table 4(b).

7.2.2 Sensitivity with Space Overhead

We evaluated the sensitivity of SRCMap energy savings with the amount of over-provisioned space to store volume working sets. Figure 13 depicts the average power consumption of the entire storage system (i.e., all eight volumes) across a 24 hour interval as the amount of over-provisioned space is varied as a percentage of the total

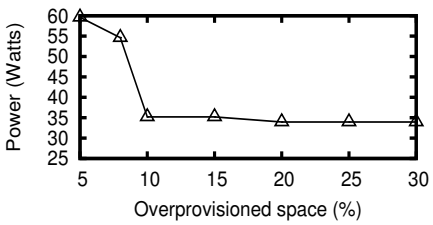


Figure 13: **Sensitivity to over-provisioned space.**

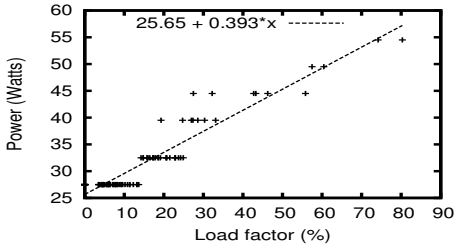


Figure 14: **Energy proportionality with load.**

storage space for the load level $L0$. We observe that SRCMap is able to deliver most of its energy savings with 10% space over-provisioning and all savings with 20%. Hence, we conclude that SRCMap can deliver power savings with minimal replica space.

7.2.3 Energy Proportionality

Our next experiment evaluates the degree of energy proportionality to the total load on the storage system delivered by SRCMap. For this experiment, we examined the power consumption within each 2-hour consolidation interval across the 24-hour duration for each of the five load estimation levels L0 through L4, giving us 60 data points. Further, we created a few higher load levels below $L0$ to study energy proportionality at high load as well. Each data point is characterized by an average power consumption value and a *load factor* value which is the observed average IOPS load as a percentage of the estimated IOPS capacity (based on the load estimation level) across all the volumes. Figure 14 presents the power consumption at each load factor. Even though the load factor is a continuous variable, power consumption levels in SRCMap are discrete. One may note that SRCMap can only vary one volume at a time and hence the different power-performance levels in SRCMap differ by one physical volume. We do observe that SRCMap is able to achieve close to N-level proportionality for a system with N-volumes, demonstrating a step-wise linear increase in power levels with increasing load.

7.3 Resource overhead of SRCMap

The primary resource overhead in SRCMap is the memory used by the *Replica Metadata (map)* of the *Replica manager*. This memory overhead depends on the size of the replica space maintained on each volume for storing both working-sets and off-loaded writes. We maintain a per-block map entry, which consists of 5 bytes to point to the current active replica. 4 additional bytes keep what replicas contain the last data version and 4 more bytes are used to handle the I/Os absorbed in the replica-space write buffer, making a total of 13 bytes for each entry in the map. If N is the number of volumes of size S with $R\%$ space to store replicas, then the worst-case memory consumption is approximately equal to the *map* size, expressed as $\frac{N \times S \times R \times 13}{2^{12}}$. For a storage virtualization manager that manages 10 volumes of total size 10TB, each with a replica space allocation of 100GB (10% over-provisioning), the memory overhead is only 3.2GB, easily affordable for a high-end storage virtualization manager.

8 Conclusions and Future Work

In this work, we have proposed and evaluated SRCMap, a storage virtualization solution for energy-proportional storage. SRCMap establishes the feasibility of an energy proportional storage system with fully flexible dynamic storage consolidation along the lines of server consolidation where any virtual machine can be migrated to any physical server in the cluster. SRCMap is able to meet all the desired goals of fine-grained energy proportionality, low space overhead, reliability, workload shift adaptation, and heterogeneity support.

Our work opens up several new directions for further research. Some of the most important modeling and optimization solutions that will improve a system like SRCMap are (i) new models that capture the performance impact of storage consolidation, (ii) investigating the use of workload correlation between logical volumes during consolidation, and (iii) optimizing the scheduling of replica synchronization to minimize impact on foreground I/O.

Acknowledgments

We would like to thank the anonymous reviewers of this paper for their insightful feedback and our shepherd Hakim Weatherspoon for his generous help with the final version of the paper. We are also grateful to Eric Johnson for providing us access to collect block level traces from production servers at FIU. This work was supported in part by the NSF grants CNS-0747038 and IIS-0534530 and by DoE grant DE-FG02-06ER25739.

References

[1] Jens Axboe. blktrace user guide, February 2007.

[2] Luiz André Barroso and Urs Hölzle. The case for energy proportional computing. In *IEEE Computer*, 2007.

[3] Luiz André Barroso and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of War ehouse-Scale Machines*. Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers, May 2009.

[4] Norman Bobroff, Andrzej Kochut, and Kirk Beaty. Dynamic placement of virtual machines for managing sla violations. In *IEEE Conf. Integrated Network Management*, 2007.

[5] D. Colarelli and D. Grunwald. Massive arrays of idle disks for storage archives. In *High Performance Networking and Computing Conference*, 2002.

[6] HP Corporation. Hp storageworks san virtualization services platform: Overview & features. http://h18006.www1.hp.com/products/storage /software/sanvr/index.html.

[7] EMC Corporation. EMC Invista. http://www.emc.com/products/software/invista/invista.jsp.

[8] Lakshmi Ganesh, Hakim Weatherspoon, Mahesh Balakrishnan, and Ken Birman. Optimizing power consumption in large scale storage systems. In *HotOS*, 2007.

[9] K. Greenan, D. Long, E. Miller, T. Schwarz, and J. Wylie. A spin-up saved is energy earned: Achieving power-efficient, erasure-coded storage. In *HotDep*, 2008.

[10] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Franke. Drpm: Dynamic speed control for power management in server class disks. In *ISCA*, 2003.

[11] S. Gurumurthi, M. R. Stan, and S. Sankar. Using intradisk parallelism to build energy-efficient storage systems. In *IEEE MICRO Top Picks*, 2009.

[12] IBM Corporation. Ibm system storage san volume controller. http://www-03.ibm.com/systems/storage/software/virtualization/svc/.

[13] IDC. Virtualization across the enterprise, Nov 2006.

[14] Patricia Kim and Mike Suk. Ramp load/unload technology in hard disk drives. *Hitachi Global Storage Technologies White Paper*, 2007.

[15] H. Lee, K. Lee, and S. Noh. Augmenting raid with an ssd for energy relief. In *HotPower*, 2008.

[16] Andrew W. Leung, Shankar Pasupathy, Garth Goodson, and Ethan L. Miller. Measurement and analysis of large-scale network file systemworkloads. In *Usenix ATC*, 2008.

[17] L. Lu and P. Varman. Diskgroup: Energy efficient disk layout for raid1 systems. In *IEEE NAS*, 2007.

[18] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. In *Usenix FAST*, 2008.

[19] Network Appliance, Inc. NetApp V-Series for Heterogeneous Storage Environments. http://media.netapp.com/documents/v-series.pdf.

[20] E. Pinheiro and R. Bianchini. Energy conservation techniques for disk array-based servers. In *ICS*, 2006.

[21] E. Pinheiro, R. Bianchini, and C. Dubnicki. Exploiting redundancy to conserve energy in storage systems. In *SIGMETRICS*, 2006.

[22] Control power and cooling for data center efficiency HP thermal logic technology. An HP Bladesystem innovation primer. http://h71028.www7.hp.com/erc/downloads/4aa0-5820enw.pdf, 2006.

[23] Mark W. Storer, Kevin M. Greenan, Ethan L. Miller, and Kaladhar Voruganti. Pergamum: Replacing tape with energy efficient, reliable disk-based archival storage. In *Usenix FAST*, 2008.

[24] Niraj Tolia, Zhikui Wang, Manish Marwah, Cullen Bash, Parthasarathy Ranganathan, and Xiaoyun Zhu. Delivering Energy Proportionality with Non Energy-Proportional Systems – Optimizing the Ensemble. In *HotPower '08: Workshop on Power Aware Computing and Systems*. ACM, December 2008.

[25] Luis Useche, Jorge Guerra, Medha Bhadkamkar, Mauricio Alarcon, and Raju Rangaswami. Exces: External caching in energy saving storage systems. In *HPCA*, 2008.

[26] A. Verma, P. Ahuja, and A. Neogi. pMapper: Power and migration cost aware application placement in virtualized systems. In *Middleware*, 2008.

[27] A. Verma, G. Dasgupta, T. Nayak, P. De, and R. Kothari. Server workload analysis for power minimization using consolidation. In *Usenix ATC*, 2009.

[28] J. Wang, X. Yao, and H. Zhu. Exploiting in-memory and on-disk redundancy to conserve energy in storage systems. In *IEEE Tran. on Computers*, 2008.

[29] Wattsup Corporation. Watts up? PRO Meter. https://www.wattsupmeters.com/secure/products.php?pn=0, 2009.

[30] C. Weddle, M. Oldham, J. Qian, A.-I. A. Wang, P. Reiher, and G. Kuenning. Paraid: a gear-shifting power-aware raid. In *Usenix FAST*, 2007.

[31] Q. Zhu, Z. Chen, L. Tan, Y. Zhou, K. Keeton, and J. Wilkes. Hibernator: helping disk arrays sleep through the winter. In *SOSP*, 2005.

A Appendix

A.1 Proof of Theorem 1

Proof: Note that the algorithm always selects the source nodes with the highest outgoing edge weight. Hence, it suffices to show that the outgoing edge weight of a source node equals (or is proportional to) the probability of it requiring a replica target on an active disk. Observe that the ordering property on weights holds in the first iteration of the algorithm as the outgoing edge weight for each $mdisk$ is the probability of it being spun down (or requiring a replica target). We argue that the re-calibration step ensures that the *Ordering property* holds inductively for all subsequent iterations.

Assuming the property holds for the m^{th} iteration, consider the $(m+1)^{th}$ iteration of the algorithm. We classify all source nodes into three categories: (i) $mdisks$ with P_i lower than the P_{m+1}, (ii) $mdisks$ with P_i higher than P_{m+1} but with no replicas assigned to targets, and (iii) $mdisks$ with P_i higher than P_{m+1} but with replicas assigned already. Note that for the first and second category of $mdisks$, the outgoing edge weights are equal to their initial values and hence their probability of their being spun down is same as the edge weights. For the third category, we restrict attention to $mdisks$ with only one replica copy, while observing that the argument holds for the general case as well. Assume that the $mdisk$ S_i has replica placed on $mdisk$ T_j. Observe then that the re-calibration property ensures that the current weight of edge $w_{i,j}$ is P_iP_j, which equals the probability that both S_i and T_j are spun down. Note also that S_i would require an active target other than T_j if T_j is also spun down, and hence the likelihood of S_i requiring a replica target (amongst active disks) is precisely P_iP_j. Hence, the ordering property holds for the $(m+1)^{th}$ iteration as well. ■

Membrane: Operating System Support for Restartable File Systems

Swaminathan Sundararaman, Sriram Subramanian, Abhishek Rajimwale,
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Michael M. Swift
Computer Sciences Department, University of Wisconsin, Madison

Abstract

We introduce Membrane, a set of changes to the operating system to support restartable file systems. Membrane allows an operating system to tolerate a broad class of file system failures and does so while remaining transparent to running applications; upon failure, the file system restarts, its state is restored, and pending application requests are serviced as if no failure had occurred. Membrane provides transparent recovery through a lightweight logging and checkpoint infrastructure, and includes novel techniques to improve performance and correctness of its fault-anticipation and recovery machinery. We tested Membrane with ext2, ext3, and VFAT. Through experimentation, we show that Membrane induces little performance overhead and can tolerate a wide range of file system crashes. More critically, Membrane does so with little or no change to existing file systems thus improving robustness to crashes without mandating intrusive changes to existing file-system code.

1 Introduction

Operating systems crash. Whether due to software bugs [8] or hardware bit-flips [22], the reality is clear: large code bases are brittle and the smallest problem in software implementation or hardware environment can lead the entire monolithic operating system to fail.

Recent research has made great headway in operating-system crash tolerance, particularly in surviving device driver failures [9, 10, 13, 14, 20, 31, 32, 37, 40]. Many of these approaches achieve some level of fault tolerance by building a *hard wall* around OS subsystems using address-space based isolation and microrebooting [2, 3] said drivers upon fault detection. For example, Nooks (and follow-on work with Shadow Drivers) encapsulate device drivers in their own protection domain, thus making it challenging for errant driver code to overwrite data in other parts of the kernel [31, 32]. Other approaches are similar, using variants of microkernel-based architectures [7, 13, 37] or virtual machines [10, 20] to isolate drivers from the kernel.

Device drivers are not the only OS subsystem, nor are they necessarily where the most important bugs reside. Many recent studies have shown that *file systems* contain a large number of bugs [5, 8, 11, 25, 38, 39]. Perhaps this is not surprising, as file systems are one of the largest and most complex code bases in the kernel. Further, file systems are still under active development, and new ones are introduced quite frequently. For example, Linux has many established file systems, including ext2 [34], ext3 [35], reiserfs [27], and still there is great interest in next-generation file systems such as Linux ext4 and btrfs. Thus, file systems are large, complex, and under development, the perfect storm for numerous bugs to arise.

Because of the likely presence of flaws in their implementation, it is critical to consider how to recover from file system crashes as well. Unfortunately, we cannot directly apply previous work from the device-driver literature to improving file-system fault recovery. File systems, unlike device drivers, are extremely *stateful*, as they manage vast amounts of both in-memory and persistent data; making matters worse is the fact that file systems spread such state across many parts of the kernel including the page cache, dynamically-allocated memory, and so forth. On-disk state of the file system also needs to be consistent upon restart to avoid any damage to the stored data. Thus, when a file system crashes, a great deal more care is required to recover while keeping the rest of the OS intact.

In this paper, we introduce *Membrane*, an operating system framework to support lightweight, stateful recovery from file system crashes. During normal operation, Membrane logs file system operations, tracks file system objects, and periodically performs lightweight checkpoints of file system state. If a file system crash occurs, Membrane parks pending requests, cleans up existing state, restarts the file system from the most recent checkpoint, and replays the in-memory operation log to restore the state of the file system. Once finished with recovery, Membrane begins to service application requests again; applications are unaware of the crash and restart except for a small performance blip during recovery.

Membrane achieves its performance and robustness through the application of a number of novel mechanisms. For example, a *generic checkpointing mechanism* enables low-cost snapshots of file system-state that serve as recovery points after a crash with minimal support from existing file systems. A *page stealing* technique greatly reduces logging overheads of write operations, which would otherwise increase time and space overheads. Finally, an intricate *skip/trust unwind protocol* is applied to carefully unwind in-kernel threads through both the crashed file

system and kernel proper. This process restores kernel state while preventing further file-system-induced damage from taking place.

Interestingly, file systems already contain many explicit error checks throughout their code. When triggered, these checks crash the operating system (e.g., by calling panic) after which the file system either becomes unusable or unmodifiable. Membrane leverages these explicit error checks and invokes recovery instead of crashing the file system. We believe that this approach will have the propaedeutic side-effect of encouraging file system developers to add a higher degree of integrity checking in order to fail quickly rather than run the risk of further corrupting the system. If such faults are transient (as many important classes of bugs are [21]), crashing and quickly restarting is a sensible manner in which to respond to them.

As performance is critical for file systems, Membrane only provides a lightweight fault detection mechanism and does not place an address-space boundary between the file system and the rest of the kernel. Hence, it is possible that some types of crashes (e.g., wild writes [4]) will corrupt kernel data structures and thus prohibit complete recovery, an inherent weakness of Membrane's architecture. Users willing to trade performance for reliability could use Membrane on top of stronger protection mechanism such as Nooks [31].

We evaluated Membrane with the ext2, VFAT, and ext3 file systems. Through experimentation, we find that Membrane enables existing file systems to crash and recover from a wide range of fault scenarios (around 50 fault injection experiments). We also find that Membrane has less than 2% overhead across a set of file system benchmarks. Membrane achieves these goals with little or no intrusiveness to existing file systems: only 5 lines of code were added to make ext2, VFAT, and ext3 restartable. Finally, Membrane improves robustness with complete application transparency; even though the underlying file system has crashed, applications continue to run.

The rest of this paper is organized as follows. Section 2 places Membrane in the context of other relevant work. Sections 3 and 4 present the design and implementation, respectively, of Membrane; finally, we evaluate Membrane in Section 5 and conclude in Section 6.

2 Background

Before presenting Membrane, we first discuss previous systems that have a similar goal of increasing operating system fault resilience. We classify previous approaches along two axes: *overhead* and *statefulness*.

We classify fault isolation techniques that incur little overhead as *lightweight*, while more costly mechanisms are classified as *heavyweight*. Heavyweight mechanisms are not likely to be adopted by file systems, which have been tuned for high performance and scalability [15, 30, 1], especially when used in server environments.

We also classify techniques based on how much system state they are designed to recover after failure. Techniques that assume the failed component has little in-memory state is referred to as *stateless*, which is the case with most device driver recovery techniques. Techniques that can handle components with in-memory and even persistent storage are *stateful*; when recovering from file-system failure, stateful techniques are required.

We now examine three particular systems as they are exemplars of three previously explored points in the design space. Membrane, described in greater detail in subsequent sections, represents an exploration into the fourth point in this space, and hence its contribution.

2.1 Nooks and Shadow Drivers

The renaissance in building isolated OS subsystems is found in Swift et al.'s work on Nooks and subsequently shadow drivers [31, 32]. In these works, the authors use memory-management hardware to build an isolation boundary around device drivers; not surprisingly, such techniques incur high overheads [31]. The kernel cost of Nooks (and related approaches) is high, in this one case spending nearly $6\times$ more time in the kernel.

The subsequent shadow driver work shows how recovery can be transparently achieved by restarting failed drivers and diverting clients by passing them error codes and related tricks. However, such recovery is relatively straightforward: only a simple reinitialization must occur before reintegrating the restarted driver into the OS.

2.2 SafeDrive

SafeDrive takes a different approach to fault resilience [40]. Instead of address-space based protection, SafeDrive automatically adds assertions into device drivers. When an assert is triggered (e.g., due to a null pointer or an out-of-bounds index variable), SafeDrive enacts a recovery process that restarts the driver and thus survives the would-be failure. Because the assertions are added in a C-to-C translation pass and the final driver code is produced through the compilation of this code, SafeDrive is lightweight and induces relatively low overheads (up to 17% reduced performance in a network throughput test and 23% higher CPU utilization for the USB driver [40], Table 6.).

However, the SafeDrive recovery machinery does not handle stateful subsystems; as a result the driver will be in an initial state after recovery. Thus, while currently well-suited for a certain class of device drivers, SafeDrive recovery cannot be applied directly to file systems.

2.3 CuriOS

CuriOS, a recent microkernel-based operating system, also aims to be resilient to subsystem failure [7]. It achieves this end through classic microkernel techniques

	Heavyweight	Lightweight
Stateless	Nooks/Shadow[31, 32]* Xen[10], Minix[13, 14] L4[20], Nexus[37]	SafeDrive[40]* Singularity[19]
Stateful	CuriOS[7] EROS[29]	Membrane*

Table 1: **Summary of Approaches.** *The table performs a categorization of previous approaches that handle OS subsystem crashes. Approaches that use address spaces or full-system checkpoint/restart are too heavyweight; other language-based approaches may be lighter weight in nature but do not solve the stateful recovery problem as required by file systems. Finally, the table marks (with an asterisk) those systems that integrate well into existing operating systems, and thus do not require the widespread adoption of a new operating system or virtual machine to be successful in practice.*

(i.e., address-space boundaries between servers) with an additional twist: instead of storing session state inside a service, it places such state in an additional protection domain where it can remain safe from a buggy service. However, the added protection is expensive. Frequent kernel crossings, as would be common for file systems in data-intensive environments, would dominate performance.

As far as we can discern, CuriOS represents one of the few systems that attempt to provide failure resilience for more stateful services such as file systems; other heavyweight checkpoint/restart systems also share this property [29]. In the paper there is a brief description of an "ext2 implementation"; unfortunately it is difficult to understand exactly how sophisticated this file service is or how much work is required to recover from failures. It also seems that there is little shared state as is common in modern systems (e.g., pages in a page cache).

2.4 Summary

We now classify these systems along the two axes of overhead and statefulness, as shown in Table 1. From the table, we can see that many systems use methods that are simply too costly for file systems; placing address-space boundaries between the OS and the file system greatly increases the amount of data copying (or page remapping) that must occur and thus is untenable. We can also see that fewer lightweight techniques have been developed. Of those, we know of none that work for stateful subsystems such as file systems. Thus, there is a need for a lightweight, transparent, and stateful approach to fault recovery.

3 Design

Membrane is designed to transparently restart the affected file system upon a crash, while applications and the rest of the OS continue to operate normally. A primary challenge in restarting file systems is to correctly manage the state associated with the file system (e.g., file descriptors, locks in the kernel, and in-memory inodes and directories).

In this section, we first outline the high-level goals for our system. Then, we discuss the nature and types of faults Membrane will be able to detect and recover from. Finally, we present the three major pieces of the Membrane system: fault detection, fault anticipation, and recovery.

3.1 Goals

We believe there are five major goals for a system that supports restartable file systems.

Fault Tolerant: A large range of faults can occur in file systems. Failures can be caused by faulty hardware and buggy software, can be permanent or transient, and can corrupt data arbitrarily or be fail-stop. The *ideal* restartable file system recovers from all possible faults.

Lightweight: Performance is important to most users and most file systems have had their performance tuned over many years. Thus, adding significant overhead is not a viable alternative: a restartable file system will only be used if it has comparable performance to existing file systems.

Transparent: We do not expect application developers to be willing to rewrite or recompile applications for this environment. We assume that it is difficult for most applications to handle unexpected failures in the file system. Therefore, the restartable environment should be completely transparent to applications; applications should not be able to discern that a file-system has crashed.

Generic: A large number of commodity file systems exist and each has its own strengths and weaknesses. Ideally, the infrastructure should enable any file system to be made restartable with little or no changes.

Maintain File-System Consistency: File systems provide different crash consistency guarantees and users typically choose their file system depending on their requirements. Therefore, the restartable environment should not change the existing crash consistency guarantees.

Many of these goals are at odds with one another. For example, higher levels of fault resilience can be achieved with heavier-weight fault-detection mechanisms. Thus in designing Membrane, we explicitly make the choice to favor performance, transparency, and generality over the ability to handle a wider range of faults. We believe that heavyweight machinery to detect and recover from relatively-rare faults is not acceptable. Finally, although Membrane should be as generic a framework as possible, a few file system modifications can be tolerated.

3.2 Fault Model

Membrane's recovery does not attempt to handle all types of faults. Like most work in subsystem fault detection and recovery, Membrane best handles failures that are *transient* and *fail-stop* [26, 32, 40].

Deterministic faults, such as memory corruption, are challenging to recover from without altering file-system

code. We assume that testing and other standard code-hardening techniques have eliminated most of these bugs. Faults such as a bug that is triggered on a given input sequence could be handled by failing the particular request. Currently, we return an error (-EIO) to the requests triggering such deterministic faults, thus preventing the same fault from being triggered again and again during recovery. Transient faults, on the other hand, are caused by race conditions and other environmental factors [33]. Thus, our aim is to mainly cope with transient faults, which can be cured with recovery and restart.

We feel that many faults and bugs can be caught with lightweight hardware and software checks. Other solutions, such as extremely large address spaces [17], could help reduce the chances of wild writes causing harm by hiding kernel objects ("needles") in a much larger addressable region ("the haystack").

Recovering a stateful file system with lightweight mechanisms is especially challenging when faults are not fail-stop. For example, consider buggy file-system code that attempts to overwrite important kernel data structures. If there is a heavyweight address-space boundary between the file system and kernel proper, then such a stray write can be detected immediately; in effect, the fault becomes fail-stop. If, in contrast, there is no machinery to detect stray writes, the fault can cause further silent damage to the rest of the kernel before causing a detectable fault; in such a case, it may be difficult to recover from the fault.

We strongly believe that once a fault is detected in the file system, no aspect of the file system should be trusted: no more code should be run in the file system and its in-memory data structures should not be used.

The major drawback of our approach is that the boundary we use is soft: some file system bugs can still corrupt kernel state outside the file system and recovery will not succeed. However, this possibility exists even in systems with hardware boundaries: data is still passed across boundaries, and no matter how many integrity checks one makes, it is possible that bad data is passed across the boundary and causes problems on the other side.

3.3 Overview

The main design challenge for Membrane is to recover file-system state in a lightweight, transparent fashion. At a high level, Membrane achieves this goal as follows.

Once a fault has been detected in the file system, Membrane rolls back the state of the file system to a point in the past that it trusts: this trusted point is a consistent file-system image that was checkpointed to disk. This checkpoint serves to divide file-system operations into distinct epochs; no file-system operation spans multiple epochs.

To bring the file system up to date, Membrane replays the file-system operations that occurred after the checkpoint. In order to correctly interpret some operations, Membrane must also remember small amounts of application-visible state from before the checkpoint, such as file descriptors. Since the purpose of this replay is only to update file-system state, non-updating operations such as reads do not need to be replayed.

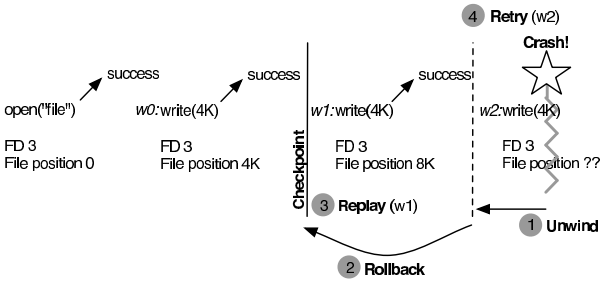


Figure 1: **Membrane Overview.** *The figure shows a file being created and written to on top of a restartable file system. Halfway through, Membrane creates a checkpoint. After the checkpoint, the application continues to write to the file; the first succeeds (and returns success to the application) and the program issues another write, which leads to a file system crash. For Membrane to operate correctly, it must (1) unwind the currently-executing write and park the calling thread, (2) clean up file system objects (not shown), restore state from the previous checkpoint, and (3) replay the activity from the current epoch (i.e., write* w1*). Once file-system state is restored from the checkpoint and session state is restored, Membrane can (4) unpark the unwound calling thread and let it reissue the write, which (hopefully) will succeed this time. The application should thus remain unaware, only perhaps noticing the timing of the third write (*w2*) was a little slow.*

Finally, to clean up the parts of the kernel that the buggy file system interacted with in the past, Membrane releases the kernel locks and frees memory the file system allocated. All of these steps are transparent to applications and require no changes to file-system code. Applications and the rest of the OS are unaffected by the fault. Figure 1 gives an example of how Membrane works during normal file-system operation and upon a file system crash.

Thus, there are three major pieces in the Membrane design. First, *fault detection* machinery enables Membrane to detect faults quickly. Second, *fault anticipation* mechanisms record information about current file-system operations and partition operations into distinct epochs. Finally, the *fault recovery* subsystem executes the recovery protocol to clean up and restart the failed file system.

3.4 Fault Detection

The main aim of fault detection within Membrane is to be lightweight while catching as many faults as possible. Membrane uses both hardware and software techniques to catch faults. The hardware support is simple: null pointers, divide-by-zero, and many other exceptions are caught by the hardware and routed to the Membrane recovery subsystem. More expensive hardware machinery, such as

address-space-based isolation, is not used.

The software techniques leverage the many checks that already exist in file system code. For example, file systems contain assertions as well as calls to `panic()` and similar functions. We take advantage of such internal integrity checking and transform calls that would crash the system into calls into our recovery engine. An approach such as that developed by SafeDrive [40] could be used to automatically place out-of-bounds pointer and other checks in the file system code.

Membrane provides further software-based protection by adding extensive parameter checking on any call from the file system into the kernel proper. These *lightweight boundary wrappers* protect the calls between the file system and the kernel and help ensure such routines are called with proper arguments, thus preventing file system from corrupting kernel objects through bad arguments. Sophisticated tools (e.g., Ballista[18]) could be used to generate many of these wrappers automatically.

3.5 Fault Anticipation

As with any system that improves reliability, there is a performance and space cost to enabling recovery when a fault occurs. We refer to this component as *fault anticipation*. Anticipation is pure overhead, paid even when the system is behaving well; it should be minimized to the greatest extent possible while retaining the ability to recover.

In Membrane, there are two components of fault anticipation. First, the *checkpointing* subsystem partitions file system operations into different *epochs* (or *transactions*) and ensures that the checkpointed image on disk represents a consistent state. Second, updates to data structures and other state are tracked with a set of *in-memory logs* and *parallel stacks*. The recovery subsystem (described below) utilizes these pieces in tandem to restart the file system after failure.

File system operations use many core kernel services (e.g., locks, memory allocation), are heavily intertwined with major kernel subsystems (e.g., the page cache), and have application-visible state (e.g., file descriptors). Careful state-tracking and checkpointing are thus required to enable clean recovery after a fault or crash.

3.5.1 Checkpointing

Checkpointing is critical because a checkpoint represents a point in time to which Membrane can safely roll back and initiate recovery. We define a checkpoint as a consistent boundary between epochs where no operation spans multiple epochs. By this definition, file-system state at a checkpoint is consistent as no file system operations are in flight.

We require such checkpoints for the following reason: file-system state is constantly modified by operations such as writes and deletes and file systems lazily write back the modified state to improve performance. As a result, at any point in time, file system state is comprised of (i) dirty pages (in memory), (ii) in-memory copies of its meta-data objects (that have not been copied to its on-disk pages), and (iii) data on the disk. Thus, the file system is in an inconsistent state until all dirty pages and meta-data objects are quiesced to the disk. For correct operation, one needs to ensure that the file system is in a consistent state at the beginning of the mount process (or the recovery process in the case of Membrane).

Modern file systems take a number of different approaches to the consistency management problem: some group updates into transactions (as in journaling file systems [12, 27, 30, 35]); others define clear consistency intervals and create snapshots (as in shadow-paging file systems [1, 15, 28]). All such mechanisms periodically create checkpoints of the file system in anticipation of a power failure or OS crash. Older file systems do not impose any ordering on updates at all (as in Linux ext2 [34] and many simpler file systems). In all cases, Membrane must operate correctly and efficiently.

The main challenge with checkpointing is to accomplish it in a lightweight and non-intrusive manner. For modern file systems, Membrane can leverage the in-built journaling (or snapshotting) mechanism to periodically checkpoint file system state; as these mechanisms atomically write back data modified within a checkpoint to the disk. To track file-system level checkpoints, Membrane only requires that these file systems explicitly notify the beginning and end of the file-system transaction (or snapshot) to it so that it can throw away the log records before the checkpoint. Upon a file system crash, Membrane uses the file system's recovery mechanism to go back to the last known checkpoint and initiate the recovery process. Note that the recovery process uses on-disk data and does not depend on the in-memory state of the file system.

For file systems that do not support any consistent-management scheme (e.g., ext2), Membrane provides a generic checkpointing mechanism at the VFS layer. Membrane's checkpointing mechanism groups several file-system operations into a single transaction and commits it atomically to the disk. A transaction is created by temporarily preventing new operations from entering into the file system for a small duration in which dirty meta-data objects are copied back to their on-disk pages and all dirty pages are marked copy-on-write. Through copy-on-write support for file-system pages, Membrane improves performance by allowing file system operations to run concurrently with the checkpoint of the *previous* epoch. Membrane associates each page with a checkpoint (or epoch) number to prevent pages dirtied in the current epoch from reaching the disk. It is important to note that the checkpointing mechanism in Membrane is implemented at the VFS layer; as a result, it can be leveraged by all file system with little or no modifications.

3.5.2 Tracking State with Logs and Stacks

Membrane must track changes to various aspects of file system state that transpired after the last checkpoint. This is accomplished with five different types of logs or stacks handling: file system operations, application-visible sessions, mallocs, locks, and execution state.

First, an in-memory *operation log (op-log)* records all state-modifying file system operations (such as open) that have taken place during the epoch or are currently in progress. The op-log records enough information about requests to enable full recovery from a given checkpoint.

Membrane also requires a small *session log (s-log)*. The s-log tracks which files are open at the beginning of an epoch and the current position of the file pointer. The op-log is not sufficient for this task, as a file may have been opened in a previous epoch; thus, by reading the op-log alone, one can only observe reads and writes to various file descriptors without the knowledge of which files such operations refer to.

Third, an in-memory *malloc table (m-table)* tracks heap-allocated memory. Upon failure, the m-table can be consulted to determine which blocks should be freed. If failure is infrequent, an implementation could ignore memory left allocated by a failed file system; although memory would be leaked, it may leak slowly enough not to impact overall system reliability.

Fourth, lock acquires and releases are tracked by the *lock stack (l-stack)*. When a lock is acquired by a thread executing a file system operation, information about said lock is pushed onto a per-thread l-stack; when the lock is released, the information is popped off. Unlike memory allocation, the exact order of lock acquires and releases is critical; by maintaining the lock acquisitions in LIFO order, recovery can release them in the proper order as required. Also note that only locks that are global kernel locks (and hence survive file system crashes) need to be tracked in such a manner; private locks internal to a file system will be cleaned up during recovery and therefore require no such tracking.

Finally, an *unwind stack (u-stack)* is used to track the execution of code in the file system and kernel. By pushing register state onto the per-thread u-stack when the file system is first called on kernel-to-file-system calls, Membrane records sufficient information to unwind threads after a failure has been detected in order to enable restart.

Note that the m-table, l-stack, and u-stack are *compensatory* [36]; they are used to compensate for actions that have already taken place and must be undone before proceeding with restart. On the other hand, both the op-log and s-log are *restorative* in nature; they are used by recovery to restore the in-memory state of the file system before continuing execution after restart.

3.6 Fault Recovery

The *fault recovery* subsystem is likely the largest subsystem within Membrane. Once a fault is detected, control is transferred to the recovery subsystem, which executes the recovery protocol. This protocol has the following phases:

Halt execution and park threads: Membrane first halts the execution of threads within the file system. Such "in-flight" threads are prevented from further execution within the file system in order to both prevent further damage as well as to enable recovery. Late-arriving threads (i.e., those that try to enter the file system after the crash takes place) are parked as well.

Unwind in-flight threads: Crashed and any other in-flight thread are unwound and brought back to the point where they are about to enter the file system; Membrane uses the u-stack to restore register values before each call into the file system code. During the unwind, any held global locks recorded on l-stack are released.

Commit dirty pages from previous epoch to stable storage: Membrane moves the system to a clean starting point at the beginning of an epoch; all dirty pages from the previous epoch are forcefully committed to disk. This action leaves the on-disk file system in a consistent state. Note that this step is not needed for file systems that have their own crash consistency mechanism.

"Unmount" the file system: Membrane consults the m-table and frees all in-memory objects allocated by the the file system. The items in the file system buffer cache (e.g., inodes and directory entries) are also freed. Conceptually, the pages from this file system in the page cache are also released mimicking an unmount operation.

"Remount" the file system: In this phase, Membrane reads the super block of the file system from stable storage and performs all other necessary work to reattach the FS to the running system.

Roll forward: Membrane uses the s-log to restore the sessions of active processes to the state they were at the last checkpoint. It then processes the op-log, replays previous operations as needed and restores the active state of the file system before the crash. Note that Membrane uses the regular VFS interface to restore sessions and to replay logs. Hence, Membrane does not require any explicit support from file systems.

Restart execution: Finally, Membrane wakes all parked threads. Those that were in-flight at the time of the crash begin execution as if they had not entered the file system; those that arrived after the crash are allowed to enter the file system for the first time, both remaining oblivious of the crash.

4 Implementation

We now present the implementation of Membrane. We first describe the operating system (Linux) environment, and then present each of the main components of Mem-

brane. Much of the functionality of Membrane is encapsulated within two components: the *checkpoint manager (CPM)* and the *recovery manager (RM)*. Each of these subsystems is implemented as a background thread and is needed during anticipation (CPM) and recovery (RM). Beyond these threads, Membrane also makes heavy use of *interposition* to track the state of various in-memory objects and to provide the rest of its functionality. We ran Membrane with ext2, VFAT, and ext3 file systems.

In implementing the functionality described above, Membrane employs three key techniques to reduce overheads and make lightweight restart of a stateful file systems feasible. The techniques are (i) *page stealing*: for low-cost operation logging, (ii) *COW-based checkpointing*: for fast in-memory partitioning of pages across epochs using copy-on-write techniques for file systems that do not support transactions, and (iii) *control-flow capture* and *skip/trust unwind protocol*: to halt in-flight threads and properly unwind in-flight execution.

4.1 Linux Background

Before delving into the details of Membrane's implementation, we first provide some background on the operating system in which Membrane was built. Membrane is currently implemented inside Linux 2.6.15.

Linux provides support for multiple file systems via the VFS interface [16], much like many other operating systems. Thus, the VFS layer presents an ideal point of interposition for a file system framework such as Membrane.

Like many systems [6], Linux file systems cache user data in a unified page cache. The page cache is thus tightly integrated with file systems and there are frequent crossings between the generic page cache and file system code.

Writes to disk are handled in the background (except when forced to disk by applications). A background I/O daemon, known as `pdflush`, wakes up, finds old and dirty pages, and flushes them to disk.

4.2 Fault Detection

There are numerous fault detectors within Membrane, each of which, when triggered, immediately begins the recovery protocol. We describe the detectors Membrane currently uses; because they are lightweight, we imagine more will be added over time, particularly as file-system developers learn to trust the restart infrastructure.

4.2.1 Hardware-based Detectors

The hardware provides the first line of fault detection. In our implementation inside Linux on x86 (64-bit) architecture, we track the following runtime exceptions: null-pointer exception, invalid operation, general protection fault, alignment fault, divide error (divide by zero), segment not present, and stack segment fault. These exception conditions are detected by the processor; software fault handlers, when run, inspect system state to determine whether the fault was caused by code executing in the file system module (i.e., by examining the faulting instruction pointer). Note that the kernel already tracks these runtime exceptions which are considered kernel errors and triggers panic as it doesn't know how to handle them. We only check if these exceptions were generated in the context of the restartable file system to initiate recovery, thus preventing kernel panic.

File System	`assert()`	`BUG()`	`panic()`
xfs	2119	18	43
ubifs	369	36	2
ocfs2	261	531	8
gfs2	156	60	0
jbd	120	0	0
jbd2	119	0	0
afs	106	38	0
jfs	91	15	6
ext4	42	182	12
ext3	16	0	11
reiserfs	1	109	93
jffs2	1	86	0
ext2	1	10	6
ntfs	0	288	2
fat	0	10	16

Table 2: **Software-based Fault Detectors.** *The table depicts how many calls each file system makes to `assert()`, `BUG()`, and `panic()` routines. The data was gathered simply by searching for various strings in the source code. A range of file systems and the ext3 journaling devices (jbd and jbd2) are included in the micro-study. The study was performed on the latest stable Linux release (2.6.26.7).*

4.2.2 Software-based Detectors

A large number of explicit error checks are extant within the file system code base; we interpose on these macros and procedures to detect a broader class of semantically-meaningful faults. Specifically, we redefine macros such as `BUG()`, `BUG_ON()`, `panic()`, and `assert()` so that the file system calls our version of said routines.

These routines are commonly used by kernel programmers when some unexpected event occurs and the code cannot properly handle the exception. For example, Linux ext2 code that searches through directories often calls `BUG()` if directory contents are not as expected; see `ext2_add_link()` where a failed scan through the directory leads to such a call. Other file systems, such as reiserfs, routinely call `panic()` when an unanticipated I/O subsystem failure occurs [25]. Table 2 presents a summary of calls present in existing Linux file systems.

In addition to those checks within file systems, we have added a set of checks across the file-system/kernel boundary to help prevent fault propagation into the kernel proper. Overall, we have added roughly 100 checks across various key points in the generic file system and memory management modules as well as in twenty or so header files. As these checks are low-cost and relatively easy to

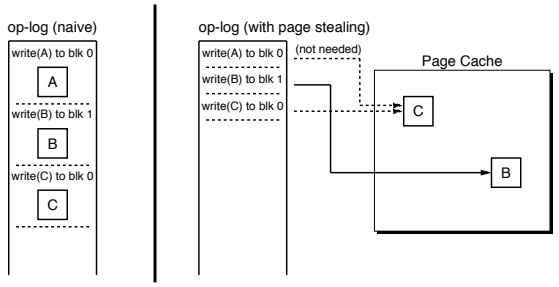


Figure 2: **Page Stealing.** *The figure depicts the op-log both with and without page stealing. Without page stealing (left side of the figure), user data quickly fills the log, thus exacting harsh penalties in both time and space overheads. With page stealing (right), only a reference to the in-memory page cache is recorded with each write; further, only the latest such entry is needed to replay the op-log successfully.*

add, we will continue to "harden" the file-system/kernel interface as our work continues.

4.3 Fault Anticipation

We now describe the fault anticipation support within the current Membrane implementation. We begin by presenting our approach to reducing the cost of operation logging via a technique we refer to as *page stealing*.

4.3.1 Low-Cost Op-Logging via Page Stealing

Membrane interposes at the VFS layer in order to record the necessary information to the op-log about file-system operations during an epoch. Thus, for any restartable file system that is mounted, the VFS layer records an entry for each operation that updates the file system state in some way.

One key challenge of logging is to minimize the amount of data logged in order to keep interpositioning costs low. A naive implementation (including our first attempt) might log all state-updating operations and their parameters; unfortunately, this approach has a high cost due to the overhead of logging write operations. For each write to the file system, Membrane has to not only record that a write took place but also log the *data* to the op-log, an expensive operation both in time and space.

Membrane avoids the need to log this data through a novel *page stealing* mechanism. Because dirty pages are held in memory before checkpointing, Membrane is assured that the most recent copy of the data is already in memory (in the page cache). Thus, when Membrane needs to replay the write, it steals the page from the cache (before it is removed from the cache by recovery) and writes the stolen page to disk. In this way, Membrane avoids the costly logging of user data. Figure 2 shows how page stealing helps in reducing the size of op-log.

When two writes to the same block have taken place, note that only the last write needs to be replayed. Earlier writes simply update the file position correctly. This strategy works because reads are not replayed (indeed, they have already completed); hence, only the current state of the file system, as represented by the last checkpoint and current op-log and s-log, must be reconstructed.

4.3.2 Other Logging and State Tracking

Membrane also interposes at the VFS layer to track all necessary session state in the s-log. There is little information to track here: simply which files are open (with their pathnames) and the current file position of each file.

Membrane also needs to track memory allocations performed by a restartable file system. We added a new allocation flag, `GFP_RESTARTABLE`, in Membrane. We also provide a new header file to include in file-system code to append `GFP_RESTARTABLE` to all memory allocation call. This enables the memory allocation module in the kernel to record the necessary per-file-system information into the m-table and thus prepare for recovery.

Tracking lock acquisitions is also straightforward. As we mentioned earlier, locks that are private to the file system will be ignored during recovery, and hence need not be tracked; only global locks need to be monitored. Thus, when a thread is running in the file system, the instrumented lock function saves the lock information in the thread's private l-stack for the following locks: the global kernel lock, super-block lock, and the inode lock.

Finally, Membrane must also track register state across certain code boundaries to unwind threads properly. To do so, Membrane wraps all calls from the kernel into the file system; these wrappers push and pop register state, return addresses, and return values onto and off of the u-stack.

4.3.3 COW-based Checkpointing

Our goal of checkpointing was to find a solution that is lightweight and works correctly despite the lack of transactional machinery in file systems such as Linux ext2, many UFS implementations, and various FAT file systems; these file systems do not include journaling or shadow paging to naturally partition file system updates into transactions.

One could implement a checkpoint using the following strawman protocol. First, during an epoch, prevent dirty pages from being flushed to disk. Second, at the end of an epoch, checkpoint file-system state by first halting file system activity and then forcing all dirty pages to disk. At this point, the on-disk state would be consistent. If a file-system failure occurred during the next epoch, Membrane could rollback the file system to the beginning of the epoch, replay logged operations, and thus recover the file system.

The obvious problem with the strawman is performance: forcing pages to disk during checkpointing makes checkpointing slow, which slows applications. Further,

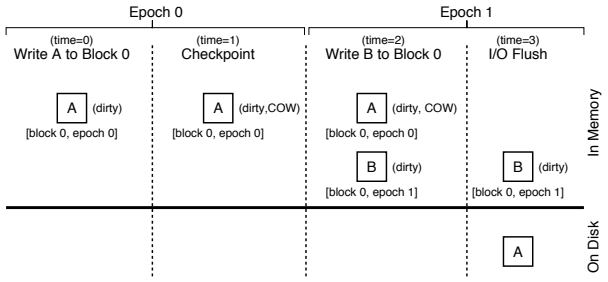


Figure 3: **COW-based Checkpointing.** *The picture shows what happens during COW-based checkpointing. At time=0, an application writes to block 0 of a file and fills it with the contents "A". At time=1, Membrane performs a checkpoint, which simply marks the block copy-on-write. Thus, Epoch 0 is over and a new epoch begins. At time=2, block 0 is over-written with the new contents "B"; the system catches this overwrite with the COW machinery and makes a new in-memory page for it. At time=3, Membrane decides to flush the previous epoch's dirty pages to disk, and thus commits block 0 (with "A" in it) to disk.*

update traffic is bunched together and must happen during the checkpoint, instead of being spread out over time; as is well known, this can reduce I/O performance [23].

Our lightweight checkpointing solution instead takes advantage of the page-table support provided by modern hardware to partition pages into different epochs. Specifically, by using the protection features provided by the page table, the CPM implements a *copy-on-write-based checkpoint* to partition pages into different epochs. This COW-based checkpoint is simply a lightweight way for Membrane to partition updates to disk into different epochs. Figure 3 shows an example on how COW-based checkpointing works.

We now present the details of the checkpoint implementation. First, at the time of a checkpoint, the checkpoint manager (CPM) thread wakes and indicates to the *session manager* (SM) that it intends to checkpoint. The SM parks new VFS operations and waits for in-flight operations to complete; when finished, the SM wakes the CPM so that it can proceed.

The CPM then walks the lists of dirty objects in the file system, starting at the superblock, and finds the dirty pages of the file system. The CPM marks these kernel pages *copy-on-write*; further updates to such a page will induce a copy-on-write fault and thus direct subsequent writes to a new copy of the page. Note that the copy-on-write machinery is present in many systems, to support (among other things) fast address-space copying during process creation. This machinery is either implemented within a particular subsystem (e.g., file systems such as ext3cow [24], WAFL [15] manually create and track their COW pages) or inbuilt in the kernel for application pages. To our knowledge, copy-on-write machinery is not available for kernel pages. Hence, we explicitly added support for copy-on-write machinery for kernel pages in Membrane; thereby avoiding extensive changes to file systems to support COW machinery.

The CPM then allows these pages to be written to disk (by tracking a checkpoint number associated with the page), and the background I/O daemon (`pdflush`) is free to write COW pages to disk at its leisure during the next epoch. Checkpointing thus groups the dirty pages from the previous epoch and allows only said modifications to be written to disk during the next epoch; newly dirtied pages are held in memory until the complete flush of the previous epoch's dirty pages.

There are a number of different policies that can be used to decide when to checkpoint. An ideal policy would likely consider a number of factors, including the time since last checkpoint (to minimize recovery time), the number of dirty blocks (to keep memory pressure low), and current levels of CPU and I/O utilization (to perform checkpointing during relatively-idle times). Our current policy is simpler, and just uses time (5 secs) and a dirty-block threshold (40MB) to decide when to checkpoint. Checkpoints are also initiated when an application forces data to disk.

4.4 Fault Recovery

We now describe the last piece of our implementation which performs fault recovery. Most of the protocol is implemented by the recovery manager (RM), which runs as a separate thread. The most intricate part of recovery is how Membrane gains control of threads after a fault occurs in the file system and the unwind protocol that takes place as a result. We describe this component of recovery first.

4.4.1 Gaining Control with Control-Flow Capture

The first problem encountered by recovery is how to gain control of threads already executing within the file system. The fault that occurred (in a given thread) may have left the file system in a corrupt or unusable state; thus, we would like to stop all other threads executing in the file system as quickly as possible to avoid any further execution within the now-untrusted file system.

Membrane, through the RM, achieves this goal by immediately marking all code pages of the file system as non-executable and thus ensnaring other threads with a technique that we refer as *control-flow capture*. When a thread that is already within the file system next executes an instruction, a trap is generated by the hardware; Membrane handles the trap and then takes appropriate action to unwind the execution of the thread so that recovery can proceed after all these threads have been unwound. File systems in Membrane are inserted as loadable kernel modules, this ensures that the file system code is in a 4KB page and not part of a large kernel page which

could potentially be shared among different kernel modules. Hence, it is straightforward to transparently identify code pages of file systems.

4.4.2 Intertwined Execution and The Skip/Trust Unwind Protocol

Unfortunately, unwinding a thread is challenging, as the file system interacts with the kernel in a tightly-coupled fashion. Thus, it is not uncommon for the file system to call into the kernel, which in turn calls into the file system, and so forth. We call such execution paths *intertwined*.

Intertwined code puts Membrane into a difficult position. Ideally, Membrane would like to unwind the execution of the thread to the beginning of the first kernel-to-file-system call as described above. However, the fact that (non-file-system) kernel code has run complicates the unwinding; kernel state will *not* be cleaned up during recovery, and thus any state modifications made by the kernel must be undone before restart.

For example, assume that the file system code is executing (e.g., in function `f1()`) and calls into the kernel (function `k1()`); the kernel then updates kernel-state in some way (e.g., allocates memory or grabs locks) and then calls back into the file system (function `f2()`); finally, `f2()` returns to `k1()` which returns to `f1()` which completes. The tricky case arises when `f2()` crashes; if we simply unwound execution naively, the state modifications made while in the kernel would be left intact, and the kernel could quickly become unusable.

To overcome this challenge, Membrane employs a careful *skip/trust unwind protocol*. The protocol *skips* over file system code but *trusts* the kernel code to behave reasonable in response to a failure and thus manage kernel state correctly. Membrane coerces such behavior by carefully arranging the return value on the stack, mimicking an error return from the failed file-system routine to the kernel; the kernel code is then allowed to run and clean up as it sees fit. We found that the Linux kernel did a good job of checking return values from the file-system function and in handling error conditions. In places where it did not (12 such instances), we explicitly added code to do the required check.

In the example above, when the fault is detected in `f2()`, Membrane places an error code in the appropriate location on the stack and returns control immediately to `k1()`. This trusted kernel code is then allowed to execute, hopefully freeing any resources that it no longer needs (e.g., memory, locks) before returning control to `f1()`. When the return to `f1()` is attempted, the control-flow capture machinery again kicks into place and enables Membrane to unwind the remainder of the stack. A real example from Linux is shown in Figure 4.

Throughout this process, the u-stack is used to capture the necessary state to enable Membrane to unwind properly. Thus, both when the file system is first entered as well as any time the kernel calls into the file system, wrapper functions push register state onto the u-stack; the values are subsequently popped off on return, or used to skip back through the stack during unwind.

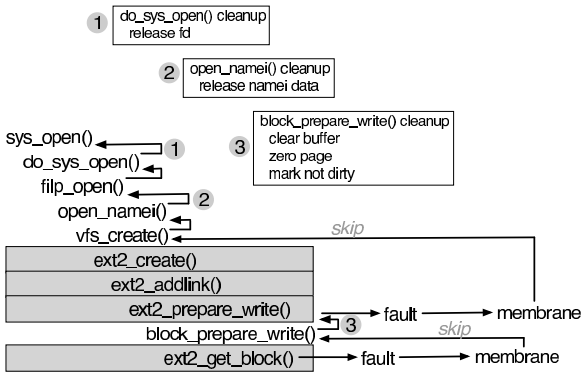


Figure 4: **The Skip/Trust Unwind Protocol.** *The figure depicts the call path from the* `open()` *system call through the ext2 file system. The first sequence of calls (through* `vfs_create()`*) are in the generic (trusted) kernel; then the (untrusted) ext2 routines are called; then ext2 calls back into the kernel to prepare to write a page, which in turn may call back into ext2 to get a block to write to. Assume a fault occurs at this last level in the stack; Membrane catches the fault, and skips back to the last trusted kernel routine, mimicking a failed call to* `ext2_get_block()`*; this routine then runs its normal failure recovery (marked by the circled "3" in the diagram), and then tries to return again. Membrane's control-flow capture machinery catches this and then* skips *back all the way to the last trusted kernel code (*`vfs_create`*), thus mimicking a failed call to* `ext2_create()`*. The rest of the code unwinds with Membrane's interference, executing various cleanup code along the way (as indicated by the circled 2 and 1).*

4.4.3 Other Recovery Functions

There are many other aspects of recovery which we do not discuss in detail here for sake of space. For example, the RM must orchestrate the entire recovery protocol, ensuring that once threads are unwound (as described above), the rest of the recovery protocol to unmount the file system, free various objects, remount it, restore sessions, and replay file system operations recorded in the logs, is carried out. Finally, after recovery, RM allows the file system to begin servicing new requests.

4.4.4 Correctness of Recovery

We now discuss the correctness of our recovery mechanism. Membrane throws away the corrupted in-memory state of the file system immediately after the crash. Since faults are fail-stop in Membrane, on-disk data is never corrupted. We also prevent any new operation from being issued to the file system while recovery is being performed. The file-system state is then reverted to the last known

checkpoint (which is guaranteed to be consistent). Next, successfully completed op-logs are replayed to restore the file-system state to the crash time. Finally, the unwound processes are allowed to execute again.

Non-determinism could arise while replaying the completed operations. The order recorded in op-logs need not be the same as the order executed by the scheduler. This new execution order could potentially pose a problem while replaying completed write operations as applications could have observed the modified state (via *read*) before the crash. On the other hand, operations that modify the file-system state (such as create, unlink, etc.) would not be a problem as conflicting operations are resolved by the file system through locking.

Membrane avoids non-deterministic replay of completed write operations through page stealing. While replaying completed operations, Membrane reads the final version of the page from the page cache and re-executes the write operation by copying the data from it. As a result, write operations while being replayed will end up with the same final version no matter what order they are executed. Lastly, as the in-flight operations have not returned back to the application, Membrane allows the scheduler to execute them in arbitrary order.

5 Evaluation

We now evaluate Membrane in the following three categories: transparency, performance, and generality. All experiments were performed on a machine with a 2.2 GHz Opteron processor, two 80GB WDC disks, and 2GB of memory running Linux 2.6.15. We evaluated Membrane using ext2, VFAT, and ext3. The ext3 file system was mounted in data journaling mode in all the experiments.

5.1 Transparency

We employ fault injection to analyze the transparency offered by Membrane in hiding file system crashes from applications. The goal of these experiments is to show the inability of current systems in hiding faults from application and how using Membrane can avoid them.

Our injection study is quite targeted; we identify places in the file system code where faults may cause trouble, and inject faults there, and observe the result. These faults represent transient errors from three different components: virtual memory (e.g., kmap, d_alloc_anon), disks (e.g., write_full_page, sb_bread), and kernel-proper (e.g., clear_inode, iget). In all, we injected 47 faults in different code paths in three file systems. We believe that many more faults could be injected to highlight the same issue.

Table 3 presents the results of our study. The caption explains how to interpret the data in the table. In all experiments, the operating system was always usable after fault injection (not shown in the table). We now discuss our major observations and conclusions.

		ext2			ext2+ boundary			ext2+ Membrane		
ext2_Function	**Fault**	**How Detected?**	**Application?**	**FS:Consistent?**	**How Detected?**	**Application?**	**FS:Consistent?**	**How Detected?**	**Application?**	**FS:Consistent?**
create	null-pointer	o	×	×	o	×	×	d	√	√
create	mark_inode_dirty	o	×	×	o	×	×	d	√	√
writepage	write_full_page	o	×	√	d	s	×	d	√	√
writepages	write_full_page	o	×	√[a]	d	s	×	d	√	√
free_inode	mark_buffer_dirty	o	×	×	o[b]	×	×	d	√	√
mkdir	d_instantiate	o	×	×	d	s	√	d	√	√
get_block	map_bh	o	×	√[a]	o[b]	×	×	d	√	√
readdir	page_address	G	×	×	G	×	×	d	√	√
get_page	kmap	o	×	√	o[b]	×	√	d	√	√
get_page	wait_page_locked	o	×	√	o[b]	×	√	d	√	√
get_page	read_cache_page	o	×	√	o	×	√	d	√	√
lookup	iget	o	×	√	o[b]	×	√	d	√	√
add_nondir	d_instantiate	o	×	×	d	e	√	d	√	√
find_entry	page_address	G	×	√	G[b]	×	√	d	√	√
symlink	null-pointer	o	×	×	o	×	√	d	√	√
rmdir	null-pointer	o	×	√	o	×	√	d	√	√
empty_dir	page_address	G	×	√	G	×	√	d	√	√
make_empty	grab_cache_page	o	×	√	o[b]	×	×	d	√	√
commit_chunk	unlock_page	o	×	√	d	e	×	d	√	√
readpage	mpage_readpage	o	×	√	i	×	√	d	√	√
		vfat			**vfat+ boundary**			**vfat+ Membrane**		
vfat_Function	**Fault**									
create	null-pointer	o	×	×	o	×	×	d	√	√
create	d_instantiate	o	×	×	o	×	×	d	√	√
writepage	blk_write_fullpage	o	×	√[a]	d	s	×	d	√	√
mkdir	d_instantiate	o	×	√	d	s	√	d	√	√
rmdir	null-pointer	o	×	√	o	×	√	d	√	√
lookup	d_find_alias	o	×	√	d	e	√	d	√	√
get_entry	sb_bread	o	×	√	o	×	√	d	√	√
get_block	map_bh	o	×	√[a]	o	×	√[a]	d	√	√
remove_entries	mark_buffer_dirty	o	×	√[a]	d	s	×	d	√	√
write_inode	mark_buffer_dirty	o	×	√[a]	d	s	√	d	√	√
clear_inode	is_bad_inode	o	×	√[a]	d	s	√	d	√	√
get_dentry	d_alloc_anon	o	×	√[a]	o[b]	×	×	d	√	√
readpage	mpage_readpage	o	×	√	o	×	√[a]	d	√	√
		ext3			**ext3+ boundary**			**ext3+ Membrane**		
ext3_Function	**Fault**									
create	null-pointer	o	×	×	o	×	√	d	√	√
get_blk_handle	bh_result	o	×	×	d	s	×	d	√	√
follow_link	nd_set_link	o	×	√[a]	d	e	√	d	√	√
mkdir	d_instantiate	o	×	×	d	s	√	d	√	√
symlink	null-pointer	o	×	×	d	×	√	d	√	√
readpage	mpage_readpage	o	×	√[a]	d	×	√[a]	d	√	√
add_nondir	d_instantiate	o	×	√	o	×	√	d	√	√
prepare_write	blk_prepare_write	o	×	√	i	e	√	d	√	√
read_blk_bmap	sb_bread	o	×	√	o	×	√	d	√	√
new_block	dquot_alloc_blk	o	×	√	o	×	√	d	√	√
readdir	null-pointer	o	×	×	o	×	√[a]	d	√	√
file_write	file_aio_write	G	×	√	i	e	√	d	√	√
free_inode	clear_inode	o	×	×	o	×	√	d	√	√
new_inode	null-pointer	o	×	√	i	×	√[a]	d	√	√

Table 3: **Fault Study.** *The table shows the results of fault injections on the behavior of Linux ext2, VFAT and ext3. Each row presents the results of a single experiment, and the columns show (in left-to-right order): which routine the fault was injected into, the nature of the fault, how/if it was detected, how it affected the application, whether the file system was consistent after the fault, and whether the file system was usable. Various symbols are used to condense the presentation. For detection, "o": kernel oops; "G": general protection fault; "i": invalid opcode; "d": fault detected, say by an assertion. For application behavior, "×": application killed by the OS; "√": application continued operation correctly; "s": operation failed but application ran successfully (silent failure); "e": application ran and returned an error. Footnotes: [a] - file system usable, but un-unmountable; [b] - late oops or fault, e.g., after an error code was returned.*

Benchmark	ext2	ext2+ Membrane	ext3	ext3+ Membrane	VFAT	VFAT+ Membrane
Seq. read	17.8	17.8	17.8	17.8	17.7	17.7
Seq. write	25.5	25.7	56.3	56.3	18.5	20.2
Rand. read	163.2	163.5	163.2	163.2	163.5	163.6
Rand. write	20.3	20.5	65.5	65.5	18.9	18.9
create	34.1	34.1	33.9	34.3	32.4	34.0
delete	20.0	20.1	18.6	18.7	20.8	21.0

Table 4: **Microbenchmarks.** *This table compares the execution time (in seconds) for various benchmarks for restartable versions of ext2, ext3, VFAT (on Membrane) against their regular versions on the unmodified kernel. Sequential read/writes are 4 KB at a time to a 1-GB file. Random reads/writes are 4 KB at a time to 100 MB of a 1-GB file. Create/delete copies/removes 1000 files each of size 1MB to/from the file system respectively. All workloads use a cold file-system cache.*

Benchmark	ext2	ext2+ Membrane	ext3	ext3+ Membrane	VFAT	VFAT+ Membrane
Sort	142.2	142.6	152.1	152.5	146.5	146.8
OpenSSH	28.5	28.9	28.7	29.1	30.1	30.8
PostMark	46.9	47.2	478.2	484.1	43.1	43.8

Table 5: **Macrobenchmarks.** *The table presents the performance (in seconds) of different benchmarks running on both standard and restartable versions of ext2, VFAT, and ext3. The sort benchmark (CPU intensive) sorts roughly 100MB of text using the command-line sort utility. For the OpenSSH benchmark (CPU+I/O intensive), we measure the time to copy, untar, configure, and make the OpenSSH 4.51 source code. PostMark (I/O intensive) parameters are: 3000 files (sizes 4KB to 4MB), 60000 transactions, and 50/50 read/append and create/delete biases.*

First, we analyzed the vanilla versions of the file systems on standard Linux kernel as our base case. The results are shown in the leftmost result column in Table 3. We observed that Linux does a poor job in recovering from the injected faults; most faults (around 91%) triggered a kernel "oops" and the application (i.e., the process performing the file system operation that triggered the fault) was always killed. Moreover, in one-third of the cases, the file system was left unusable, thus requiring a reboot and repair (*fsck*).

Second, we analyzed the usefulness of fault detection without recovery by hardening the kernel and file-system boundary through parameter checks. The second result column (denoted by +boundary) of Table 3 shows the results. Although assertions detect the bad argument passed to the kernel proper function, in the majority of the cases, the returned error code was not handled properly (or propagated) by the file system. The application was always killed and the file system was left inconsistent, unusable, or both.

Finally, we focused on file systems surrounded by Membrane. The results of the experiments are shown in the rightmost column of Table 3; faults were handled, applications did not notice faults, and the file system remained in a consistent and usable state.

In summary, even in a limited and controlled set of fault injection experiments, we can easily realize the usefulness of Membrane in recovering from file system crashes. In a standard or hardened environment, a file system crash is almost always visible to the user and the process performing the operation is killed. Membrane, on detecting a file system crash, transparently restarts the file system and leaves it in a consistent and usable state.

5.2 Performance

To evaluate the performance of Membrane, we run a series of both microbenchmark and macrobenchmark workloads where ext2, VFAT, and ext3 are run in a standard environment and within the Membrane framework.

Tables 4 and 5 show the results of our microbenchmark and macrobenchmark experiments respectively. From the tables, one can see that the performance overheads of our prototype are quite minimal; in all cases, the overheads were between 0% and 2%.

Data (MB)	Recovery time (ms)
10	12.9
20	13.2
40	16.1

(a)

Open Sessions	Recovery time (ms)
200	11.4
400	14.6
800	22.0

(b)

Log Records	Recovery time (ms)
1K	15.3
10K	16.8
100K	25.2

(c)

Table 6: **Recovery Time.** *Tables a, b, and c show recovery time as a function of dirty pages (at checkpoint), s-log, and op-log respectively. Dirty pages are created by copying new files. Open sessions are created by getting handles to files. Log records are generated by reading and seeking to arbitrary data inside multiple files. The recovery time was 8.6ms when all three states were empty.*

Recovery Time. Beyond baseline performance under no crashes, we were interested in studying the performance of Membrane during recovery. Specifically, how long does it take Membrane to recover from a fault? This metric is particularly important as high recovery times may be noticed by applications.

We measured the recovery time in a controlled environment by varying the amount of state kept by Membrane and found that the recovery time grows sub-linearly with the amount of state and is only a few milliseconds in all the cases. Table 6 shows the result of varying the amount of state in the s-log, op-log and the number of dirty pages from the previous checkpoint.

We also ran microbenchmarks and forcefully crashed ext2, ext3, and VFAT file systems during execution to measure the impact in application throughput inside Membrane. Figure 5 shows the results for performing recovery during the random-read microbenchmark for the ext2 file system. From the figure, we can see that Membrane restarts the file system within 10ms from the point of crash. Subsequent read operations are slower than the regular case because the indirect blocks, that were cached by the file system, are thrown away at recovery time in our current prototype and have to be read back again after recovery (as shown in the graph).

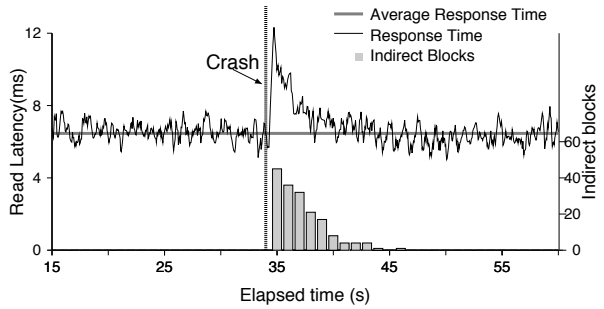


Figure 5: **Recovery Overhead.** *The figure shows the overhead of restarting ext2 while running random-read microbenchmark. The x axis represents the overall elapsed time of the microbenchmark in seconds. The primary y axis contains the execution time per read operation as observed by the application in milliseconds. A file-system crash was triggered at 34s, as a result the total elapsed time increased from 66.5s to 67.1s. The secondary y axis contains the number of indirect blocks read by the ext2 file system from the disk per second.*

In summary, both micro and macrobenchmarks show that the fault anticipation in Membrane almost comes for free. Even in the event of a file system crash, Membrane restarts the file system within a few milliseconds.

5.3 Generality

We chose ext2, VFAT, and ext3 to evaluate the generality of our approach. ext2 and VFAT were chosen for their lack of crash consistency machinery and for their completely different on-disk layout. ext3 was selected for its journaling machinery that provides better crash consistency guarantees than ext2. Table 7 shows the code changes required in each file system.

File System	Added	Modified
ext2	4	0
VFAT	5	0
ext3	1	0
JBD	4	0

Individual File-system Changes

Components	No Checkpoint		With Checkpoint	
	Added	Modified	Added	Modified
FS	1929	30	2979	64
MM	779	5	867	15
Arch	0	0	733	4
Headers	522	6	552	6
Module	238	0	238	0
Total	**3468**	**41**	**5369**	**89**

Kernel Changes

Table 7: **Implementation Complexity.** *The table presents the code changes required to transform a ext2, VFAT, ext3, and vanilla Linux 2.6.15 x86_64 kernel into their restartable counterparts. Most of the modified lines indicate places where vanilla kernel did not check/handle errors propagated by the file system. As our changes were non-intrusive in nature, none of existing code was removed from the kernel.*

From the table, we can see that the file system specific changes required to work with Membrane are minimal. For ext3, we also added 4 lines of code to JBD to notify the beginning and the end of transactions to the checkpoint manager, which could then discard the operation logs of the committed transactions. All of the additions were straightforward, including adding a new header file to propagate the `GFP_RESTARTABLE` flag and code to write back the free block/inode/cluster count when the `write_super` method of the file system was called. No modification (or deletions) of existing code were required in any of the file systems.

In summary, Membrane represents a generic approach to achieve file system restartability; existing file systems can work with Membrane with minimal changes of adding a few lines of code.

6 Conclusions

File systems fail. With Membrane, failure is transformed from a show-stopping event into a small performance issue. The benefits are many: Membrane enables file-system developers to ship file systems sooner, as small bugs will not cause massive user headaches. Membrane similarly enables customers to install new file systems, knowing that it won't bring down their entire operation.

Membrane further encourages developers to harden their code and catch bugs as soon as possible. This fringe benefit will likely lead to more bugs being triggered in the field (and handled by Membrane, hopefully); if so, diagnostic information could be captured and shipped back to the developer, further improving file system robustness.

We live in an age of imperfection, and software imperfection seems a fact of life rather than a temporary state of affairs. With Membrane, we can learn to embrace that imperfection, instead of fearing it. Bugs will still arise, but those that are rare and hard to reproduce will remain where they belong, automatically "fixed" by a system that can tolerate them.

7 Acknowledgments

We thank the anonymous reviewers and Dushyanth Narayanan (our shepherd) for their feedback and comments, which have substantially improved the content and presentation of this paper. We also thank Haryadi Gunawi for his insightful comments.

This material is based upon work supported by the National Science Foundation under the following grants: CCF-0621487, CNS-0509474, CNS-0834392, CCF-0811697, CCF-0811697, CCF-0937959, as well as by generous donations from NetApp, Sun Microsystems, and Google.

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] Jeff Bonwick and Bill Moore. ZFS: The Last Word in File Systems. http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf, 2007.

[2] George Candea and Armando Fox. Crash-Only Software. In *The Ninth Workshop on Hot Topics in Operating Systems (HotOS IX)*, Lihue, Hawaii, May 2003.

[3] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot – A Technique for Cheap Recovery. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 31–44, San Francisco, California, December 2004.

[4] John Chapin, Mendel Rosenblum, Scott Devine, Tirthankar Lahiri, Dan Teodosiu, and Anoop Gupta. Hive: Fault Containment for Shared-Memory Multiprocessors. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, Copper Mountain Resort, Colorado, December 1995.

[5] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An Empirical Study of Operating System Errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 73–88, Banff, Canada, October 2001.

[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] Francis M. David, Ellick M. Chan, Jeffrey C. Carlyle, and Roy H. Campbell. CuriOS: Improving Reliability through Operating System Structure. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, California, December 2008.

[8] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 57–72, Banff, Canada, October 2001.

[9] Ulfar Erlingsson, Martin Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software Guards for System Address Spaces. In *Proceedings of the 7th USENIX OSDI*, pages 6–6, 2006.

[10] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe Hardware Access with the Xen Virtual Machine Monitor. In *Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, 2004.

[11] Haryadi S. Gunawi, Cindy Rubio-Gonzalez, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Ben Liblit. EIO: Error Handling is Occasionally Correct. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, pages 207–222, San Jose, California, February 2008.

[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] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Construction of a Highly Dependable Operating System. In *Proceedings of the 6th European Dependable Computing Conference*, October 2006.

[14] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Failure Resilience for Device Drivers. In *Proceedings of the 2007 IEEE International Conference on Dependable Systems and Networks*, pages 41–50, June 2007.

[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] Steve R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *Proceedings of the USENIX Summer Technical Conference (USENIX Summer '86)*, pages 238–247, Atlanta, Georgia, June 1986.

[17] E. Koldinger, J. Chase, and S. Eggers. Architectural Support for Single Address Space Operating Systems. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, Boston, Massachusetts, October 1992.

[18] Nathan P. Kropp, Philip J. Koopman, and Daniel P. Siewiorek. Automated Robustness Testing of Off-the-Shelf Software Components. In *Proceedings of the 28th International Symposium on Fault-Tolerant Computing (FTCS-28)*, Munich, Germany, June 1998.

[19] James Larus. The Singularity Operating System. Seminar given at the University of Wisconsin, Madison, 2005.

[20] J. LeVasseur, V. Uhlig, J. Stoess, and S. Gotz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Proceedings of the 6th USENIX OSDI*, 2004.

[21] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from Mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*, Seattle, Washington, March 2008.

[22] Dejan Milojicic, Alan Messer, James Shau, Guangrui Fu, and Alberto Munoz. Increasing Relevance of Memory Hardware Errors: A Case for Recoverable Programming Models. In *9th ACM SIGOPS European Workshop 'Beyond the PC: New Challenges for the Operating System'*, Kolding, Denmark, September 2000.

[23] Jeffrey C. Mogul. A Better Update Policy. In *Proceedings of the USENIX Summer Technical Conference (USENIX Summer '94)*, Boston, Massachusetts, June 1994.

[24] Zachary Peterson and Randal Burns. Ext3cow: a time-shifting file system for regulatory compliance. *Trans. Storage*, 1(2):190–212, 2005.

[25] 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.

[26] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating Bugs As Allergies. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, Brighton, United Kingdom, October 2005.

[27] Hans Reiser. ReiserFS. www.namesys.com, 2004.

[28] 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.

[29] J. S. Shapiro and N. Hardy. EROS: A Principle-Driven Operating System from the Ground Up. *IEEE Software*, 19(1), January/February 2002.

[30] 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.

[31] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, New York, October 2003.

[32] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 1–16, San Francisco, California, December 2004.

[33] Nisha Talagala and David Patterson. An Analysis of Error Behaviour in a Large Storage System. In *The IEEE Workshop on Fault Tolerance in Parallel and Distributed Systems*, San Juan, Puerto Rico, April 1999.

[34] Theodore Ts'o. http://e2fsprogs.sourceforge.net, June 2001.

[35] Theodore Ts'o and Stephen Tweedie. Future Directions for the Ext2/3 Filesystem. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, Monterey, California, June 2002.

[36] W. Weimer and George C. Necula. Finding and Preventing Run-time Error-Handling Mistakes. In *The 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*, Vancouver, Canada, October 2004.

[37] Dan Williams, Patrick Reynolds, Kevin Walsh, Emin Gün Sirer, and Fred B. Schneider. Device Driver Safety Through a Reference Validation Mechanism. In *Proceedings of the 8th USENIX OSDI*, 2008.

[38] 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.

[39] 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.

[40] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques. In *Proceedings of the 7th USENIX OSDI*, Seattle, Washington, November 2006.