USENIX

Proceedings of FAST '11: 9th USENIX Conference on File and Storage Technologies

San Jose, CA, USA February 15–17, 2011

conference

proceedings

FAST '11: 9th USENIX Conference on File and Storage Technologies

San Jose, CA, USA
February 15–17, 2011

Sponsored by

USENIX

in cooperation with
ACM SIGOPS

ISBN 978-1-931971-82-9

USENIX Association

Proceedings of FAST ’11: 9th USENIX Conference on File and Storage Technologies

February 15–17, 2011
San Jose, CA, USA

Conference Organizers

Program Co-Chairs

Greg Ganger, *Carnegie Mellon University*
John Wilkes, *Google*

Program Committee

Marcos K. Aguilera, *Microsoft Research*
Cristiana Amza, *University of Toronto*
John Bent, *Los Alamos National Lab*
Jeff Chase, *Duke University*
Jeff Hammerbacher, *Cloudera*
Steve Hand, *University of Cambridge*
Wilson Hsieh, *Google*
Arkady Kanevsky, *VMware*
Christos Karamanolis, *VMware*
Michael A. Kozuch, *Intel Labs Pittsburgh*
Carlos Maltzahn, *University of California, Santa Cruz*
Arif Merchant, *Google*
Brian Noble, *University of Michigan*
James Plank, *University of Tennessee*
Benjamin Reed, *Yahoo! Research*
Ohad Rodeh, *IBM Almaden Research Center*
Rob Ross, *Argonne National Lab*
Karsten Schwan, *Georgia Institute of Technology*
Keith Smith, *NetApp*
Eno Thereska, *Microsoft Research*
Cristian Ungureanu, *NEC Labs*
Elizabeth Varki, *University of New Hampshire*
Andrew Warfield, *University of British Columbia*
Hakim Weatherspoon, *Cornell University*

Tutorial Chair

David Pease, *IBM Almaden Research Center*

Steeering Committee

Remzi H. Arpaci-Dusseau, *University of Wisconsin—Madison*
Randal Burns, *Johns Hopkins University*
Greg Ganger, *Carnegie Mellon University*
Garth Gibson, *Carnegie Mellon University and Panasas*
Peter Honeyman, *CITI, University of Michigan, Ann Arbor*
Kimberly Keeton, *HP Labs*
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
Akshat Aranya
Rico D'Amore
Mikkel Hagen
Tim Harris
Allen Hubbe
Erik Kruus
Mark Lillibridge
Grant Mackey
Adam Manzanares
Dutch T. Meyer
Sigfredo Nin
James Nunez
Milo Polte
Bianca Schroeder
Mustafa Uysal
Adam Villa

FAST '11: 9th USENIX Conference on File and Storage Technologies
February 15–17, 2011
San Jose, CA, USA

Wednesday, February 16

Deduplication

Specializing Storage

Flash

The Disk Ain't Dead

Thursday, February 17

Scaling Well

Making Things Right

Flash the Second

Message from the Program Co-Chairs

Dear Colleagues,

We welcome you to the 9th USENIX Conference on File and Storage Technologies (FAST '11). 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 deduplication. It displays the breadth of storage systems research with work on storage in cloud computing and low-power sensor networks. It also contains significant contributions to the core of the field, including disk systems, trace and benchmarking studies, and RAID.

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 mix of papers from universities, from companies, and from collaborations between the two arenas.

FAST '11 received 74 submissions (slightly lower than in previous years), from which 20 papers were selected, for an acceptance rate of 27%. Every paper received at least three reviews from PC members, and every paper discussed in the second round—including all the accepted ones—received at least 6 reviews from PC members. For a few papers, additional external reviews were solicited.

The review process was conducted online over two months and at a program committee meeting held in Mountain View, CA, in November 2010. We again used Eddie Kohler's HotCRP software to handle paper submissions, reviews, PC discussion, and notifications. We used a two-round process that relied almost entirely on our excellent program committee for reviews. In the first round, each paper was assigned to three PC members. We then culled about half the papers and commissioned an additional three PC reviews for the remaining papers. Additional reviews were obtained for a few controversial papers—those with high variance in reviews. Twenty-four of the twenty-six PC members attended the PC meeting in person, and one other by video conferencing. The quality of the conversation at that meeting contributed significantly to the quality of the decisions that we were able to make.

We would like to thank everybody who contributed to assembling this program. First and foremost, we are indebted to all of the authors who submitted papers to FAST '11. We had a good body of high-quality work from which to select our program. We would also like to thank the attendees of FAST '11 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 the contributions of USENIX and the USENIX staff, who make everything else about assembling a conference program easy. The USENIX staff dealt with innumerable issues large and small and provided outstanding technical and emotional support. They are a delight to work with, 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 a huge amount of work. Each PC member completed 13 to 15 in-depth reviews of 11+ page papers, as well as participating in the discussions and helping out in other roles such as session chairs, poster/WiP selection, and choosing the best papers.

We look forward to seeing you in San Jose!

Greg Ganger, *Carnegie Mellon University*
John Wilkes, *Google*
Program Co-Chairs

A Study of Practical Deduplication

Dutch T. Meyer[*†] and William J. Bolosky[*]
[*]Microsoft Research and [†]The University of British Columbia
{dmeyer@cs.ubc.edu, bolosky@microsoft.com}

Abstract

We collected file system content data from 857 desktop computers at Microsoft over a span of 4 weeks. We analyzed the data to determine the relative efficacy of data deduplication, particularly considering whole-file versus block-level elimination of redundancy. We found that whole-file deduplication achieves about three quarters of the space savings of the most aggressive block-level deduplication for storage of live file systems, and 87% of the savings for backup images. We also studied file fragmentation finding that it is not prevalent, and updated prior file system metadata studies, finding that the distribution of file sizes continues to skew toward very large unstructured files.

1 Introduction

File systems often contain redundant copies of information: identical files or sub-file regions, possibly stored on a single host, on a shared storage cluster, or backed-up to secondary storage. Deduplicating storage systems take advantage of this redundancy to reduce the underlying space needed to contain the file systems (or backup images thereof). Deduplication can work at either the sub-file [10, 31] or whole-file [5] level. More fine-grained deduplication creates more opportunities for space savings, but necessarily reduces the sequential layout of some files, which may have significant performance impacts when hard disks are used for storage (and in some cases [33] necessitates complicated techniques to improve performance). Alternatively, whole-file deduplication is simpler and eliminates file-fragmentation concerns, though at the cost of some otherwise reclaimable storage.

Because the disk technology trend is toward improved sequential bandwidth and reduced per-byte cost with little or no improvement in random access speed, it's not clear that trading away sequentiality for space savings makes sense, at least in primary storage.

In order to evaluate the tradeoff in space savings between whole-file and block-based deduplication, we conducted a large-scale study of file system contents on desktop Windows machines at Microsoft. Our study consists of 857 file systems spanning 162 terabytes of disk over 4 weeks. It includes results from a broad cross-section of employees, including software developers, testers, management, sales & marketing, technical support, documentation writers and legal staff. We find that while block-based deduplication of our dataset can lower storage consumption to as little as 32% of its original requirements, nearly three quarters of the improvement observed could be captured through whole-file deduplication and sparseness. For four weeks of full backups, whole file deduplication (where a new backup image contains a reference to a duplicate file in an old backup) achieves 87% of the savings of block-based. We also explore the parameter space for deduplication systems, and quantify the relative benefits of sparse file support. Our study of file content is larger and more detailed than any previously published effort, which promises to inform the design of space-efficient storage systems.

In addition, we have conducted a study of metadata and data layout, as the last similar study [1] is now 4 years old. We find that the previously observed trend toward storage being consumed by files of increasing size continues unabated; half of all bytes are in files larger than 30MB (this figure was 2MB in 2000). Complicating matters, these files are in opaque unstructured formats with complicated access patterns. At the same time there are increasingly many small files in an increasingly complex file system tree.

Contrary to previous work [28], we find that file-level fragmentation is not widespread, presumably due to regularly scheduled background defragmenting in Windows [17] and the finding that a large portion of files are rarely modified (see Section 4.4.2). For more than a decade, file system designers have been warned against measuring only fresh file system installations, since aged systems can have a significantly different performance profile [28]. Our results show that this concern may no longer be relevant, at least to the extent that the aging produces file-level fragmentation. Ninety-six percent of files observed are entirely linear in the block address space. To our knowledge, this is the first large scale study of disk fragmentation in the wild.

We describe in detail the novel analysis optimizations necessitated by the size of this data set.

2 Methodology

Potential participants were selected randomly from Microsoft employees. Each was contacted with an offer to install a file system scanner on their work computer(s) in exchange for a chance to win a prize. The scanner ran autonomously during off hours once per week from September 18 – October 16, 2009. We contacted 10,500 people in this manner to reach the target study size of about 1000 users. This represents a participation rate of roughly 10%, which is smaller than the rates of 22% in similar prior studies [1, 9]. Anecdotally, many potential participants declined explicitly because the scanning process was quite invasive.

2.1 File system Scanner

The scanner first took a consistent snapshot of fixed device (non-removable) file systems with the Volume Shadow Copy Service (VSS) [20]. VSS snapshots are both file system and application consistent[1]. It then recorded metadata about the file system itself, including age, capacity, and space utilization. The scanner next processed each file in the snapshot, writing records to a log. It recorded Windows file metadata [19], including path, file name and extension, time stamps, and the file attribute flags. It recorded any retrieval and allocation pointers, which describe fragmentation and sparseness respectively. It also recorded information about the whole system, including the computer's hardware and software configuration and the time at which the defragmentation tool was last run, which is available in the Windows registry. We took care to exclude from study the pagefile, hibernation file, the scanner itself, and the VSS snapshots it created.

During the scan, we recorded the contents of each file first by breaking the file into chunks using each of two chunking algorithms (fixed block and Rabin fingerprinting [25]) with each of 4 chunk size settings (8K-64K in powers of two) and then computed and saved hashes of each chunk. We found whole file duplicates in post-processing by identifying files in which all chunks matched. In addition to reading the ordinary contents of files we also collected a separate set of scans where the files were read using the Win32 BackupRead API [16], which includes metadata about the file and would likely be the format used to store file system backups.

We used salted MD5 [26] as our hash algorithm, but truncated the result to 48 bits in order to reduce the size of the data set. The Rabin-chunked data with an 8K target chunk size had the largest number of unique hashes, somewhat more than 768M. We expect that about two thousand of those (0.0003%) are false matches due to the truncated hash.

Another process copied the log files to our server at midnight on a random night of the week to help smooth the considerable network traffic. Nevertheless, the copying process resulted in the loss of some of the scans. Because the scanner placed the results for each of the 32 parameter settings into separate files and the copying process worked at the file level, for some file systems we have results for some, but not all of the parameter settings. In particular, larger scan files tended to be partially copied more frequently than smaller ones, which may result in a bias in our data where larger file systems are more likely to be excluded. Similarly, scans with a smaller chunk size parameter resulted in larger size scan files and so were lost at a higher rate.

2.2 Post Processing

At the completion of the study the resulting data set was 4.12 terabytes compressed, which would have required considerable machine time to import into a database. As an optimization, we observed that the actual value of any unique hash (i.e., hashes of content that was not duplicated) was not useful to our analyses.

To find these unique hashes quickly we used a novel 2-pass algorithm. During the first pass we created a 2 GB Bloom filter [4] of each hash observed. During this pass, if we tried to insert a value that was already in the Bloom filter, we inserted it into a second Bloom filter of equal size. We then made a second pass through the logs, comparing each hash to the second Bloom filter only. If it was not found in the second filter, we were certain that the hash had been seen exactly once and could be omitted from the database. If it was in the filter, we concluded that either the hash value had been seen more than once, or that its entry in the filter was a collision. We recorded all of these values to the database. Thus this algorithm was sound, in that it did not impact the results by rejecting any duplicate hashes.

[1] "Application consistent" means that VSS-aware applications have an opportunity to save their state cleanly before the snapshot is taken.

However it was not complete despite being very effective, in that some non-duplicate hashes may have been added to the database even though they were not useful in the analysis. The inclusion of these hashes did not affect our results, as the later processing ignored them.

2.3 Biases and Sources of Error

The use of Windows workstations in this study is beneficial in that the results can be compared to those of similar studies [1, 9]. However, as in all data sets, this choice may introduce biases towards certain types of activities or data. For example, corporate policies surrounding the use of external software and libraries could have impacted our results.

As discussed above, the data retrieved from machines under observation was large and expensive to generate and so resulted in network timeouts at our server or aborted scans on the client side. While we took measures to limit these effects, nevertheless some amount of data never made it to the server, and more had to be discarded as incomplete records. Our use of VSS makes it possible for a user to selectively remove some portions of their file system from our study.

We discovered a rare concurrency bug in the scanning tool affecting 0.003% of files. While this likely did not affect results, we removed all files with this artifact.

Our scanner was unable to read the contents of Windows system restore points, though it could see the file metadata. We excluded these files from the deduplication analyses, but included them in the metadata analyses.

3 Redundancy in File Contents

Despite the significant declines in storage costs per GB, many organizations have seen dramatic increases in total storage system costs [21]. There is considerable interest in reducing these costs, which has given rise to deduplication techniques, both in the academic community [6] and as commercial offerings [7, 10, 14, 33]. Initially, the interest in deduplication has centered on its use in “embarrassingly compressible” scenarios, such as regular full backups [3, 8] or virtual desktops [6, 13]. However, some have also suggested that deduplication be used more widely on general purpose data sets [31].

The rest of this section seeks to provide a well-founded measure of duplication rates and compare the efficacy of different parameters and methods of deduplication. In Section 3.1 we provide a brief summary of deduplication, and in Section 3.2 we discuss the performance challenges deduplication introduces. In Section 3.3 we share observed duplication rates across a set of workstations. Finally, Section 3.4 measures duplication in the more conventional backup scenario.

3.1 Background on Deduplication

Deduplication systems decrease storage consumption by identifying distinct chunks of data with identical content. They then store a single copy of the chunk along with metadata about how to reconstruct the original files from the chunks.

Chunks may be of a predefined size and alignment, but are more commonly of variable size determined by the content itself. The canonical algorithm for variable-sized content-defined blocks is Rabin Fingerprints [25]. By deciding chunk boundaries based on content, files that contain identical content that is shifted (say because of insertions or deletions) will still result in (some) identical chunks. Rabin-based algorithms are typically configured with a minimum and maximum chunk size, as well as an expected chunk size. In all our experiments, we set the minimum and maximum parameters to 4K and 128K, respectively while we varied the expected chunk size from 8K to 64K by powers-of-two.

3.2 The Performance Impacts of Deduplication

Managing the overheads introduced by a deduplication system is challenging. Naively, each chunk’s fingerprint needs to be compared to that of all other chunks. While techniques such as caches and Bloom filters can mitigate overheads, the performance of deduplication systems remains a topic of research interest [32]. The I/O system also poses a performance challenge. In addition to the layer of indirection required by deduplication, deduplication has the effect of de-linearizing data placement, which is at odds with many data placement optimizations, particularly on hard-disk based storage where the cost for non-sequential access can be orders of magnitude greater than sequential.

Other more established techniques to reduce storage consumption are simpler and have smaller performance impact. Sparse file support exists in many file systems including NTFS [23], XFS [29], and ext4 [15] and is relatively simple to implement. In a sparse file a chunk of zeros is stored notationally by marking its existence in the metadata, removing the need to physically store it. Whole file deduplication systems, such as the Windows SIS facility [5] operate by finding entire files that

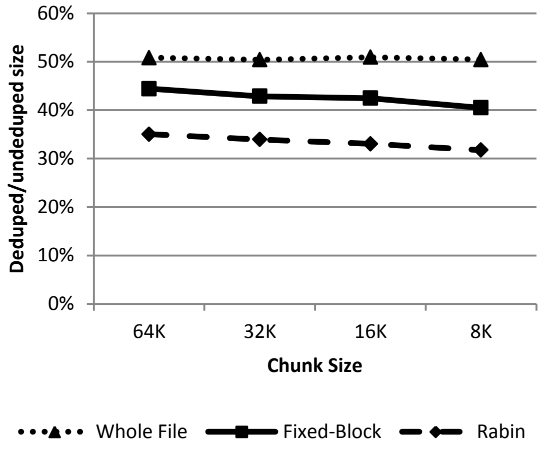


Figure 1: Deduplication vs. Chunk Size for Various Algorithms

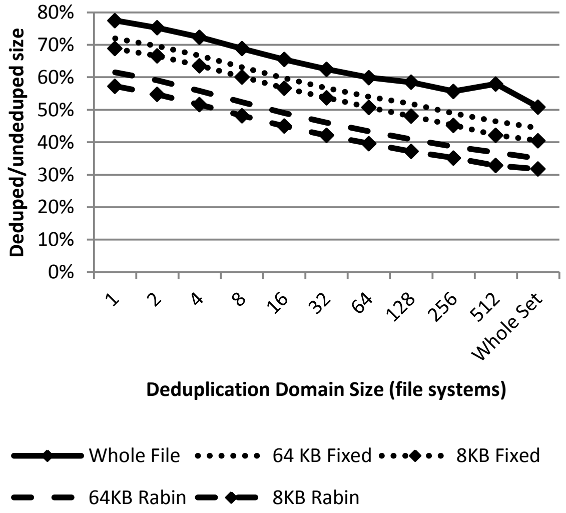


Figure 2: Deduplication vs. Deduplication Domain Size

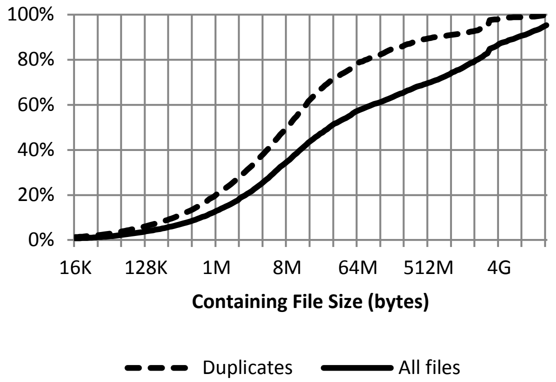


Figure 3: CDF of Bytes by Containing File Size for Whole File Duplicates and All Files

Extension	% of Duplicate Space	Mean File Size (bytes)
dll	20%	521K
lib	11%	1080K
pdb	11%	2M
<none>	7%	277K
exe	6%	572K
cab	4%	4M
msp	3%	15M
msi	3%	5M
iso	2%	436M
<a guid>	1%	604K
hxs	1%	2M
xml	1%	49K
jpg	1%	147K
wim	1%	16M
h	1%	23K

Table 1: Whole File Duplicates by Extension

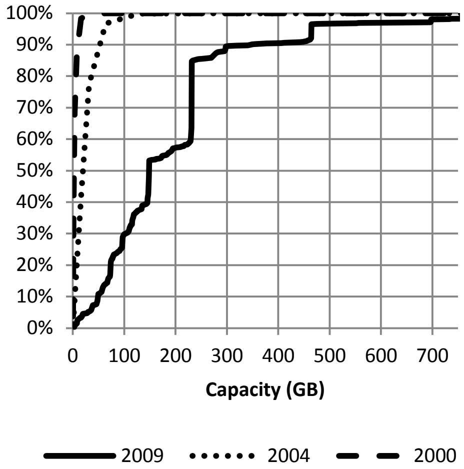


Figure 4: CDF of File System Capacity

Extension	Fixed %	Extension	Rabin %
vhd	3.6%	vhd	5.2%
pch	0.5%	lib	1.6%
dll	0.5%	obj	0.8%
pdb	0.4%	pdb	0.6%
lib	0.4%	pch	0.6%
wma	0.3%	iso	0.6%
pst	0.3%	dll	0.6%
<none>	0.3%	avhd	0.5%
avhd	0.3%	wma	0.4%
mp3	0.3%	wim	0.4%
pds	0.2%	zip	0.3%
iso	0.2%	pst	0.3%

Table 2: Non-whole File, Non-Zero Duplicate Data as a Fraction of File System Size by File Extension, 8K Fixed and Rabin Chunking

are duplicates and replacing them by copy-on-write links. Although SIS does not reduce storage consumption as much as a modern deduplication system, it avoids file allocation concerns and is far less computationally expensive than more exhaustive deduplication.

3.3 Deduplication in Primary Storage

Our data set includes hashes of data in both variable and fixed size chunks, and of varying sizes. We chose a single week (September 18, 2009) from this dataset and compared the size of all unique chunks to the total consumption observed. We had two parameters that we could vary: the deduplication algorithm/parameters and the set of file systems (called the *deduplication domain)* within which we found duplicates; duplicates in separate domains were considered to be unique contents.

The set of file systems included corresponds to the size of the file server(s) holding the machines' file systems. A value of 1 indicates deduplication running independently on each desktop machine. "Whole Set" means that all 857 file systems are stored together in a single deduplication domain. We considered all power-of-two domain sizes between 1 and 857. For domain sizes other than 1 or 857, we had to choose which file systems to include together into particular domains and which to exclude when the number of file systems didn't divide evenly by the size of the domain. We did this by using a cryptographically secure random number generator. We generated sets for each domain size ten times and report the mean of the ten runs. The standard deviation of the results was less than 2% for each of the data points, so we don't believe that we would have gained much more precision by running more trials[2].

Rather than presenting a three dimensional graph varying both parameters, we show two slices through the surface. In both cases, the y-axis shows the deduplicated file system size as a percentage of the original file system size. Figure 1 shows the effect of the chunk size parameter for the fixed and Rabin-chunked algorithms, and also for the whole file algorithm (which doesn't depend on chunk size, and so varies only slightly due to differences in the number of zeroes found and due to variations in which file systems scans copied properly; see Section 3.2). This graph assumes that all file systems are in a single deduplication domain; the shape of the curve is similar for smaller domains, through the space savings are reduced.

Figure 2 shows the effect changing the size of the deduplication domains. Space reclaimed improves roughly linearly in the log of the number of file systems in a domain. Comparing single file systems to the whole set, the effect of grouping file systems together is larger than that from the choice of chunking algorithm or chunk size, or even of switching from whole file chunking to block-based.

The most aggressive chunking algorithm (8K Rabin) reclaimed between 18% and 20% more of the total file size than did whole file deduplication. This offers weak support for block-level deduplication in primary storage. The 8K fixed block algorithm reclaimed between 10% and 11% more space than whole file. This capacity savings represents a small gain compared to the performance and complexity of introducing advanced deduplication features, especially ones with dynamically variable block sizes like Rabin fingerprinting.

Table 1 shows the top 15 file extensions contributing to duplicate content for whole file duplicates, the percentage of duplicate space attributed to files of that type, and the mean file size for each type. It was calculated using all of the file systems in a single deduplication domain. The extension marked <a guid> is a particular globally unique ID that's associated with a widely distributed software patch. This table shows that the savings due to whole file duplicates are concentrated in files containing program binaries: dll, lib, pdb, exe, cab, msp, and msi together make up 58% of the saved space.

Figure 3 shows the CDF of the bytes reclaimed by whole file deduplication and the CDF of all bytes, both by containing file size. It shows that duplicate bytes tend to be in smaller files than bytes in general. Another way of looking at this is that the very large file types (virtual hard disks, database stores, *etc*.) tend not to have whole-file copies. This is confirmed by Table 1.

Table 2 shows the amount of duplicate content not in files with whole-file duplicates by file extension as a fraction of the total file system content. It considers the whole set of file systems as a single deduplication domain, and presents results with an 8K block size using both fixed and Rabin chunking. For both algorithms, by far the largest source of duplicate data is VHD (virtual hard drive) files. Because these files are essentially disk images, it's not surprising both that they contain duplicate data and also that they rarely have whole-file duplicates. The next four file types are all compiler outputs. We speculate that they generate block-aligned duplication because they have header fields that contain, for example, timestamps but that their contents is

[2] As it was, it took about 8 machine-months to do the analyses.

otherwise deterministic in the code being compiled. Rabin chunking may find blocks of code (or symbols) that move somewhat in the file due to code changes that affect the length of previous parts of the file.

3.4 Deduplication in Backup Storage

Much of the literature on deduplication to date has relied on workloads consisting of daily full backups [32, 33]. Certainly these workloads represent the most attractive scenario for deduplication, because the content of file systems does not change rapidly. Our data set did not allow us to consider daily backups, so we considered only weekly ones.

With frequent and persistent backups, the size of historical data will quickly out-pace that of the running system. Furthermore, performance in secondary storage is less critical than in that of primary, so the reduced sequentiality of a block-level deduplicated store is of lesser concern. We considered the 483 file systems for which four continuous weeks of complete scans were available, starting with September 18, 2009, the week used for the rest of the analyses.

Our backup analysis considers each file system as a separate deduplication domain. We expect that combining multiple backups into larger domains would have a similar effect to doing the same thing for primary storage, but we did not run the analysis due to resource constraints.

In practice, some backup solutions are incremental (or differential), storing deltas between files, while others use full backups. Often, highly reliable backup policies use a mix of both, performing frequent incremental backups, with occasional full backups to limit the potential for loss due to corruption. Thus, the meaning of whole-file deduplication in a backup store is not immediately obvious. We ran the analysis as if the backups were stored as simple copies of the original file systems, except that the contents of the files was the output from the Win32 BackupRead [16] call, which includes some file metadata along with the data. For our purposes, imagine that the backup format finds whole file duplicates and stores pointers to them in the backup file. This would result in a garbage collection problem for the backup files when they're deleted, but the details of that are beyond the scope of our study and are likely to be simpler than a block-level deduplicating store.

Using the Rabin chunking algorithm with an 8K expected chunk size, block-level deduplication reclaimed 83% of the total space. Whole file deduplication, on the other hand, yielded 72%. These numbers, of course, are highly sensitive to the number of weeks of scans used in the study; it's no accident that the results were around ¾ of the space being claimed when there were four weeks of backups. However, one should not assume that because 72% of the space was reclaimed by whole file deduplication that only 3% of the bytes were in files that changed. The amount of change was larger than that, but the deduplicator found redundancy within a week as well and the two effects offset.

4 Metadata

This paper is the 3rd major metadata study of Windows desktop computers [1, 9]. This provides a unique perspective in the published literature, as we are able to track more than a decade of trends file and file system metadata. On a number of graphs, we took the lines from 2000 and 2004 from an earlier study [1] and plotted them on our graphs to make comparisons easier. Only the 2009 data is novel to this paper. Some graphs contain both CDF and histogram lines. In these graphs, the CDF should be read from the left-hand *y*-scale and the histogram from the right. We present much of our data in the form of cumulative density function plots. These plots make it easy to determine the distributions, but do not easily show the mean. Where appropriate, we list the mean of the distribution in the text.

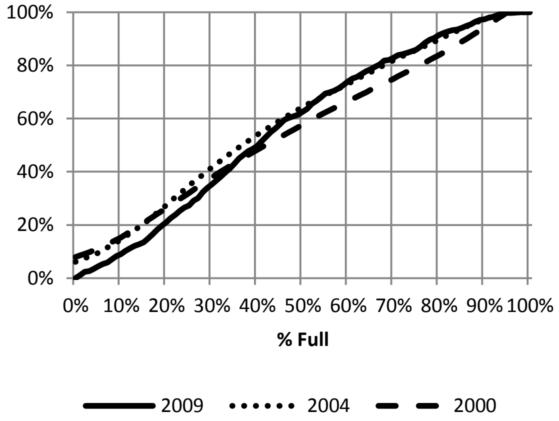


Figure 5: CDF of File Systems by Fullness

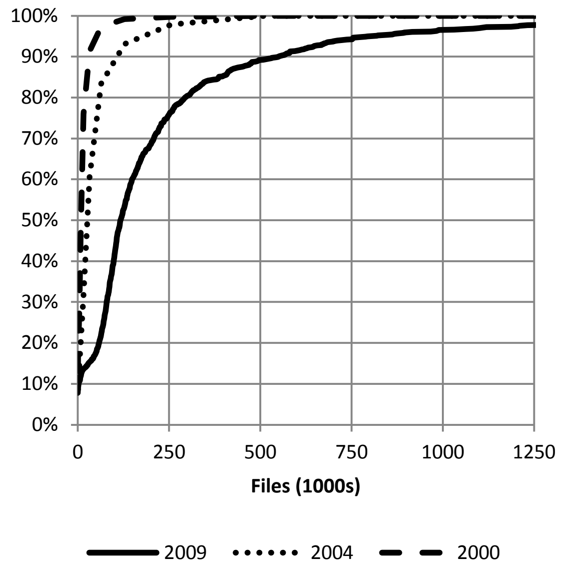


Figure 6: CDF of File Systems by Count of Files

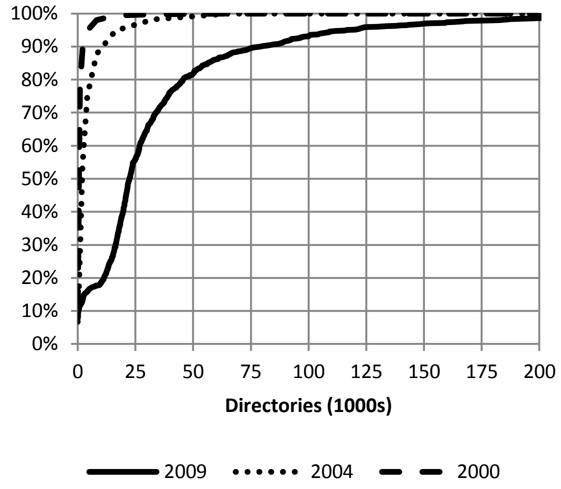


Figure 7: CDF of File Systems by Count of Directories

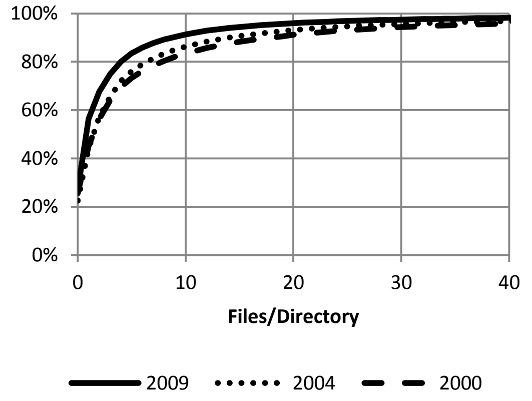


Figure 8: CDF of Directories by Count of Files

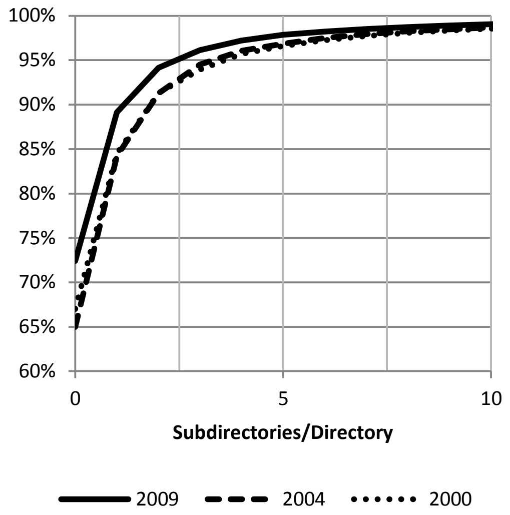


Figure 9: CDF of Directories by Count of Subdirectories

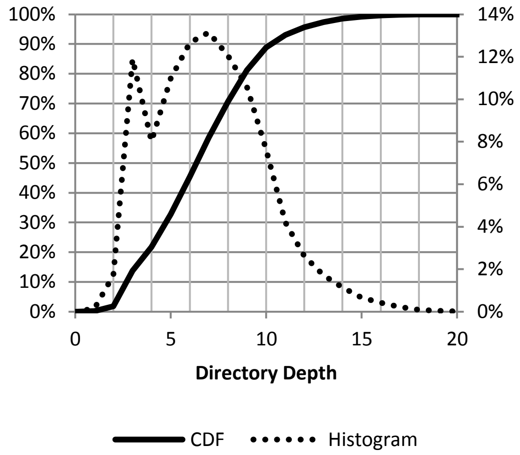


Figure 10: Files by Directory Depth

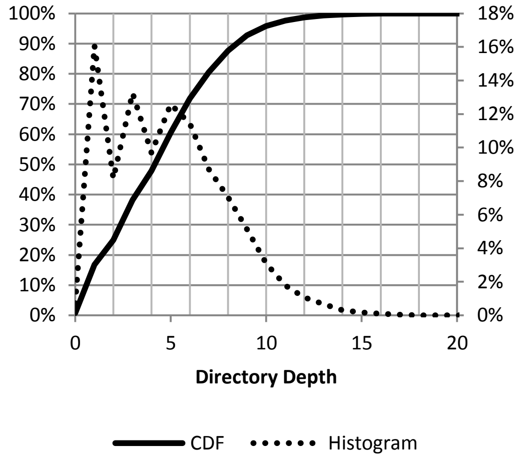


Figure 11: Bytes by Directory Depth

4.1 Physical Machines

Our data set contains scans of 857 file systems hosted on 597 computers. 59% were running Windows 7, 20% Windows Vista, 18% Windows Server 2008 and 3% Windows Server 2003. They had a mean and median physical RAM of about 4GB, and ranged from 1-10GB. 5% had 8 processors, 44% 4, 49% 2 and 3% were uniprocessors[3].

4.2 File systems

We analyze file systems in terms of their age, capacity, fullness, and the number of files and directories. We present our results, interpretations, and recommendations to designers in this section.

4.2.1 Capacity

The mean file system capacity is 194GB. Figure 4 shows a cumulative density function of the capacities of the file systems in the study. It shows a significant increase in the range of commonly observed file system sizes and the emergence of a noticeable step function in the capacities. Both of these trends follow from the approximately annual doubling of physical drive capacity. We expect that this file system capacity range will continue to increase, anchored by smaller SSDs on the left, and continuing step wise towards larger magnetic devices on the right. This will either force file systems to perform acceptably on an increasingly wide range of media, or push users towards more highly tuned special purpose file systems.

4.2.2 Utilization

Although capacity has increased by nearly two orders of magnitude since 2000, utilization of capacity has dropped only slightly, as shown in Figure 5. Mean utilization is 43%, only somewhat less than the 53% found in 2000. No doubt this is the result of both users adapting to their available space and hard drive manufacturers tracking the growth in data. The CDF shows a nearly linear relationship, with 50% of users having drives no more than 40% full, 70% at less than 60% utilization, and 90% at less than 80%. Proposals to take advantage of the unused capacity of file systems [2, 11] must be cautious that they only assume scaling of the magnitude of free space, not the relative portion of the disk that is free. System designers also must take care not to ignore the significant contingent (15%) of all users with disks more than 75% full.

4.3 File system Namespace

Recently, Murphy and Seltzer have questioned the merits of hierarchical file systems [22], based partly on the challenge of managing increasing data sizes. Our analysis shows many ways in which namespaces have become more complex. We have observed more files, more directories, and an increase in namespace depth. While a rigorous comparison of namespace organization structures is beyond the scope of this paper, the increase in namespace complexity does lend evidence to the argument that change is needed in file system organization. Both file and directory counts show a significant increase from previous years in Figures 6 and 7 respectively, with a mean of 225K files and 36K directories per file system.

The CDF in Figure 8 shows the number of files per directory. While the change is small, it is clear – even as users in 2009 have more files, they have fewer files per directory, with a mean of 6.25 files per directory.

Figure 9 shows the distribution of subdirectories per directory. Since the mean subdirectories per directory is necessarily one[4], the fact that the distribution is more skewed toward smaller sizes indicates that the directory structure is deeper with a smaller branching factor. However, the exact interpretation of this result warrants further study. It is not clear if this depth represents a conscious organization choice, is the result of users being unable effectively to organize their hierarchical data or is simply due to the design of the software that populates the tree. Figure 10 shows the histogram and CDF of files by directory depth for the 2009 data; similar results were not published in the earlier studies.

The histogram in Figure 11 shows how the utilization of storage is related to namespace depth. There is a steep decline in the number of bytes stored more than 5 levels deep in the tree. However, as we will see in Section 4.4, this does not mean the deeply nested files are unimportant. Comparing it with Figure 10 shows that files higher in the directory tree are larger than those deeper.

4.4 Files

Our analysis of files in the dataset shows distinct classes of files emerging. The frequently observed fact that most files are small and most bytes are in large files has intensified. The mean file size is now 318K, about three times what it was in 2000. Files can be classified by

[3] The total is 101% due to rounding error.

[4] Ignoring that the root directory isn't a member of any directory.

their update time as well. A large class of files is written only once (perhaps at install time).

4.4.1 File Size

In one respect, file sizes have not changed at all. The median file size remains 4K (a result that has been remarkably consistent since at least 1981 [27]), and the distribution of file sizes has changed very little since 2000. Figure 12 shows that the proportion of these small files has in fact increased with fewer files both somewhat larger and somewhat smaller than 4K. There is also an increase in larger files between 512K and 8MB.

Figure 13 shows a histogram of the total number of bytes stored in files of various sizes. A trend towards bi-modality has continued, as predicted in 2007 [1], though a third mode above 16G is now appearing. Figure 14 shows that more capacity usage has shifted to the larger files, even though there are still few such files in the system. This suggests that optimizing for large files will be increasingly important.

Viewed a different way, we can see that trends towards very large files being the principle consumers of storage have continued smoothly. As discussed in Section 4.5, this is a particular challenge because large files like VHDs have complex internal structures with difficult to predict access patterns. Semantic knowledge to exploit these structures, or file system interfaces that explicitly support them may be required to optimize for this class of data.

4.4.2 File Times

File modifications time stamps are usually updated when a file is written. Figure 15 shows a histogram and CDF of time since file modification with log scaling on the x-axis[5]. The same data with 1 month bins is plotted in Figure 16. Most files are modified between one month and a year ago, but about 20% are modified within the last month.

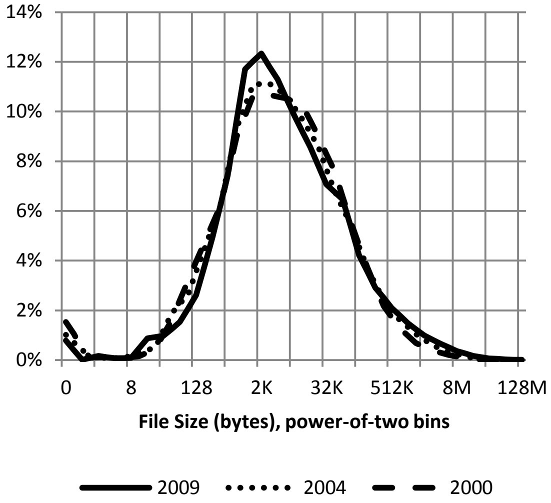


Figure 12: Histogram of Files by Size

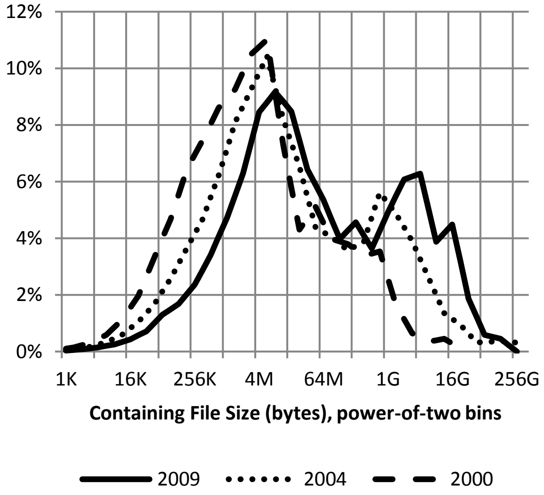


Figure 13: Histogram of Bytes by Containing File Size

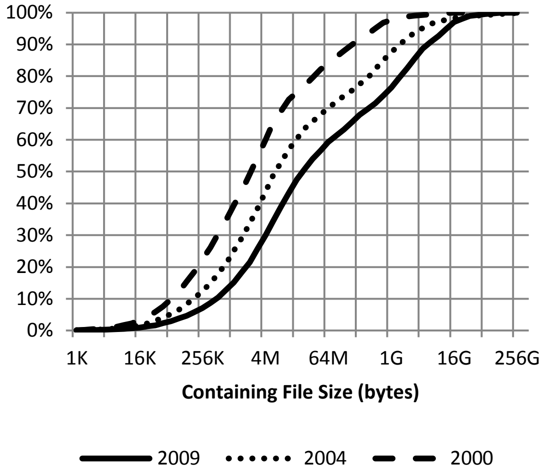


Figure 14: CDF of Bytes by Containing File Size

[5] Unlike the other combined histogram/CDF graphs, this one has both lines using the left y-axis due to a bug in the graphing package.

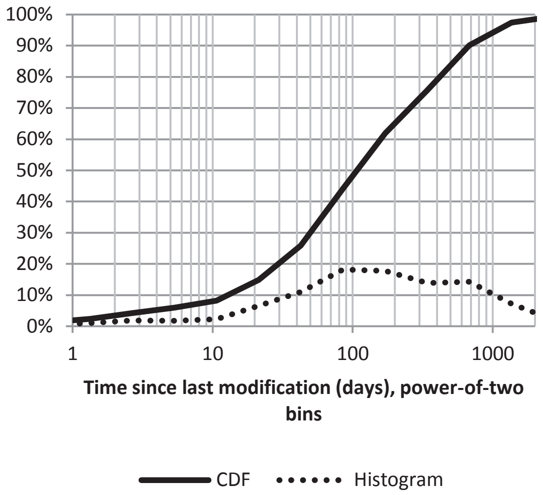


Figure 15: Time Since Last File Modification

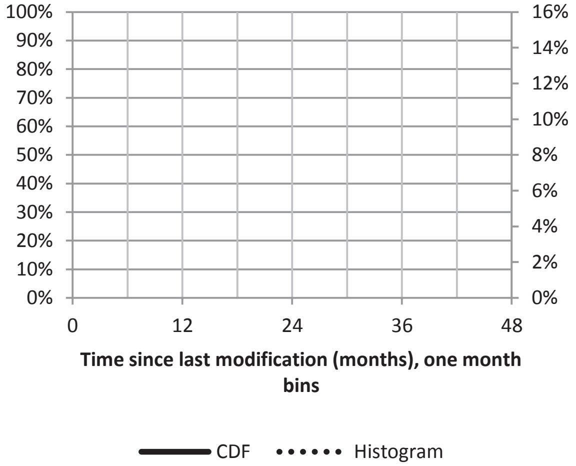


Figure 16: Time Since Last File Modification

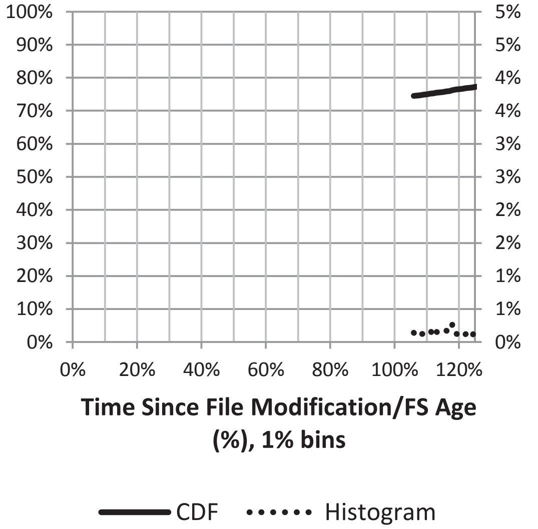


Figure 17: Time Since Last File Modification as a Fraction of File System Age.

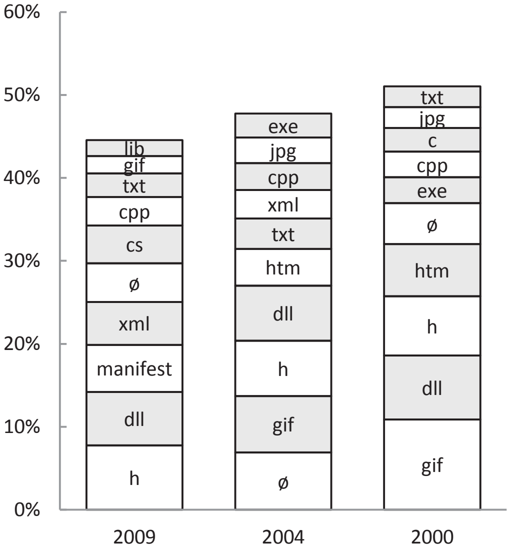


Figure 18: Popularity of Files by Extension

Figure 17 relates file modification time to the age of the file system. The *x*-axis shows the time since a file was last modified divided by the time since the file system was formatted. This range exceeds 100% because some files were created prior to installation and were subsequently copied to the file system, preserving their modification time. The spike around 100% mostly consists of files that were modified during the system installation. The area between 0% and 100% shows a relatively smooth decline, with a slight inflection around 40%.

NTFS has always supported a last access time field for files. We omit any analysis because updates to it are disabled by default as of Windows Vista [18].

4.5 Extensions

Figure 18 shows only modest change in the extensions for the most popular files. However, the extension space continues to grow. The ten most popular files extensions now account for less than 45% of the total files compared with over 50% in 2000.

Figure 19 shows the top storage consumers by file extension. Several changes are apparent here. First, there is a significant increase in storage consumed by files with no extension, which have moved from 10th place in all previous years to be the largest class of files today, replacing DLLs. VHD and ISO files are virtual disks and images for optical media. They have increased in relative size, but not as quickly as LIB files. Finally, the portion of storage space consumed by the top extensions has increased by nearly 15% from previous years.

5 On-disk Layout

The behavior and characteristics of magnetic disks continue to be a dominant concern in storage system optimization. It has been shown that file system performance changes over time, largely due to fragmentation [28]. While we have no doubt that the findings were true in 1997, our research suggests that this observation no longer holds in practice.

We measure fragmentation in our data set by recording the files' retrieval pointers, which point to NFTS's data blocks. Retrieval pointers that are non-linear indicate a fragmented file. We find such fragmentation to be rare, occurring in only 4% of files. This lack of fragmentation in Windows desktops is due to the fact that a large fraction of files are not written after they are created and due to the defragmenter, which runs weekly by default[6]. However, among files containing at least one fragment, fragments are relatively common. In fact, 25% of fragments are in files containing more than 170 fragments. The most highly fragmented files appear to be log files, which (if managed naively) may create a new fragment for each appending write.

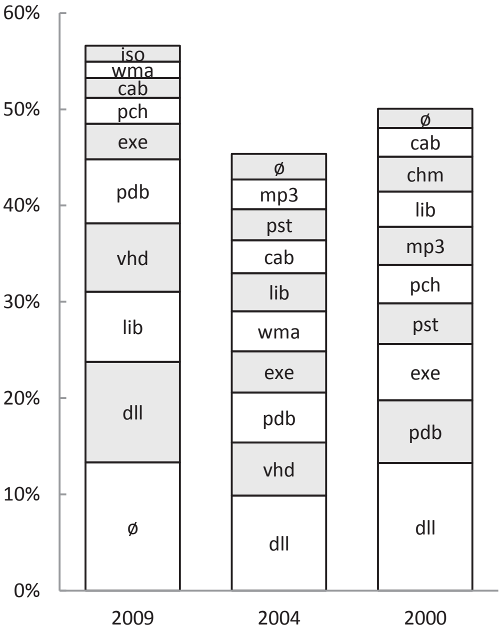


Figure 19: Bytes by File Extension

6 Related Work

Studies of live deployed system behavior and usage have long been a key component of storage systems research. Workload studies [30] are helpful in determining what file systems do in a given slice of time, but provide little guidance as to the long term contents of files or file systems. Prior file system content studies [1, 9] have considered collections of machines similar to those observed here. The most recent such study uses 7 year old data, while data from the study before it is 11 years old, which we believe justifies the file system portion of this work. However, this research also captures relevant results that the previous work does not.

Policroniades and Pratt [24] studied duplication rates using various chunking strategies on a dataset about 0.1% of the size of ours, finding little whole-file duplication and a modest difference between fixed-block and content-based chunking. Kulkarni *et al*. [12] found combining compression, eliminating duplicate identical-sized chunks and delta-encoding across multiple datasets to be effective. Their corpus was about 8GB.

We are able to track file system fragmentation and data placement, which has not been analyzed recently [28] or at large scale. We are also able to track several forms of deduplication, which is an important area of current research. Prior work has used very selective data sets usually focusing either on frequent full backups [3, 8], virtual machine images [6, 13], or simulation [10]. In the former case, data not modified between backups can be trivially deduplicated, and in the latter disk images start from a known identical storage, and diverge slowly over time. In terms of size, only the DataDomain [33] study rivals ours. It is less than half the size presented here and was for a highly self-selective group. Thus, we not only consider a more general, but also a larger dataset than comparable studies. Moreover, we include a comparison to whole-file deduplication, which has been missing in much of the deduplication research to date. Whole file deduplication is an obvious alternative to block-based deduplication because it is light-weight and as we have shown, nearly as effective at reclaiming space.

[6] This is true for all of our scans other than the 17 that came from machines running Windows Server 2003.

7 Conclusion

We studied file system data, metadata, and layout on nearly one thousand Windows file systems in a commercial environment. This new dataset contains metadata records of interest to file system designers, data content findings that will help create space efficiency techniques, and data layout information useful in the evaluation and optimization of storage systems.

We find that whole-file deduplication together with sparseness is a highly efficient means of lowering storage consumption, even in a backup scenario. It approaches the effectiveness of conventional deduplication at a much lower cost in performance and complexity. The environment we studied, despite being homogeneous, shows a large diversity in file system and file sizes. These challenges, the increase in unstructured files, and an ever-deepening and more populated namespace pose significant challenge for future file system designs. However, at least one problem – that of file fragmentation, appears to be solved, provided that a machine has periods of inactivity in which defragmentation can be run.

Acknowledgements

We would like to thank the hundreds of Microsoft employees who were willing to allow us to install software that read the entire contents of their disks, Richard Draves for helping us with the Microsoft bureaucracy, Microsoft as whole for funding and enabling this kind of research, our program committee shepherd Keith Smith and the anonymous reviewers for their guidance as well as detailed and helpful comments, and Fred Douglis for some truly last-minute comments and proof-reading.

References

[1] N. Agrawal, W. Bolosky, J. Douceur and J. Lorch. A five-year study of file-system metadata. In *Proc. 5th USENIX Conference on File and Storage Technologies*, 2007.

[2] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis. Borg: block-reorganization for self-optimizing storage systems. In *Proc. 7th USENIX Conference on File and Storage Technologies*, 2009.

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

[4] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM* 13(7): 422—426, 1970.

[5] W. Bolosky, S. Corbin, D. Goebel and J. Douceur. Single instance storage in Windows 2000. In *Proc. 4th USENIX Windows Systems Symposium*, 2000.

[6] A. Clements, I. Ahmad, M. Vilayannur, J. Li. Decentralized deduplication in SAN cluster file systems. In *Proc. USENIX Annual Technical Conference*, 2009.

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

[8] S. Dorward and S. Quinlan. Venti: A new approach to archival data storage. In *Proc. 1st USENIX Conference on File and Storage Technologies*, 2002.

[9] J. Douceur and W. Bolosky. A large-scale study of file-system contents. In *Proc. 1999 ACM SIGMETRICS International Conference on Measurement and Modelling of Computer Systems,* 1999.

[10] 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 *Proc. 7th USENIX Conference on File and Storage Technologies*, 2009.

[11] H. Huang, W. Hung, and K. G. Shin. Fs2: dynamic data replication in free disk space for improving disk performance and energy consumption. In *Proc. 20th ACM Symposium on Operating Systems Principles*, 2005.

[12] P. Kulkarni, F. Douglis, J. LaVoie, and J. Tracey. Redundancy elimination within large collections of files. In *Proc. USENIX 2004 Annual Technical Conference*, 2004.

[13] K. Jin and E. Miller. The effectiveness of deduplication on virtual machine disk images. In *Proc. SYSTOR 2009: The Israeli Experimental Systems Conference*, 2009.

[14] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble. Sparse indexing: large scale, inline deduplication using sampling and locality. In *Proc. 7th USENIX Conference on File and Storage Technologies*, 2009.

[15] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. The new ext4 filesystem: current status and future plans. In *Proc. of the Linux Symposium*, June, 2007.

[16] Microsoft Corporation. BackupRead Function. MSDN. [Online] 2010. [Cited: August 17, 2010.] http://msdn.microsoft.com/en-us/library/aa362509(VS.85).aspx.

[17] Microsoft Corporation**.** Description of the scheduled tasks in Widows Vista. *Microsoft Support.* [Online] July 8, 2010. [Cited: August 9, 2010.] http://support.microsoft.com/kb/939039.

[18] Microsoft Corporation. Disabling Last Access Time in Windows Vista to Improve NTFS Perfomance. The Storage Team Blog. [Online] 2006. [Cited November 2, 2010.] http://blogs.technet.com/b/filecab/archive/2006/11/07/disabling-last-access-time-in-windows-vista-to-improve-ntfs-performance.aspx.

[19] Microsoft Corporation. File systems. Microsoft TechNet. [Online] 2010. [Cited: August 9, 2010.] http://technet.microsoft.com/en-us/library/cc938929.aspx.

[20] Microsoft Corporation. Volume Shadow Copy Service. MSDN. [Online] 2010. [Cited August 31, 2010.] http://msdn.microsoft.com/en-us/library/bb968832(VS.85).aspx

[21] D. R. Miller. Storage Economics: Four Principles for Reducing Total Cost of Ownership. *Hitachi Corporate Web Site.* [Online] May 2009. [Cited: August 17, 2010.] http://www.hds.com/assets/pdf/four-principles-for-reducing-total-cost-of-ownership.pdf.

[22] N. Murphy and M. Seltzer. Hierarchical file systems are dead. In *Proc. 12th Workshop on Hot Topics in Operating Systems*, 2009.

[23] R. Nagar. *Windows NT File System Internals*. O'Reilly, 1997

[24] C. Policroniades and I. Pratt. Alternatives for detecting redundancy in storage systems. In *Proc. USENIX 2004 Annual Technical Conference*, 2004.

[25] M. Rabin. Fingerprinting by Random Polynomials. Harvard University Center for Research In Computing Technology Technical Report TR-CSE-03-01, 1981. Boston, MA.

[26] R. Rivest. *The MD5 Message-Digest Algorithm*. [Online] April 1992. [Cited: August 17, 2010.] http://tools.ietf.org/rfc/rfc1321.txt.

[27] Satyanarayanan, M. A study of file sizes and functional lifetimes. In *Proc. 8th ACM Symposium on Operating Systems Principles*, 1981.

[28] M. Seltzer and K. Smith. File system aging: increasing the relevance of file system benchmarks. In *Proc. 1997 ACM SIGMETRICS*, June 1997.

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

[30] W. Vogels. File system usage in windows NT 4.0. In *Proc. 17th ACM Symposium on Operating Systems Principles*, 1999.

[31] C. Ungureanu, B. Atkin, A. Aranya, S. Gokhale, S. Rago, G. Cakowski, C. Dubnicki, and A. Bohra. Hydrafs: A high-throughput file system for the Hydrastor content-addressable storage system. In *Proc. 8th USENIX Conference on File and Storage Technologies*, 2010.

[32] E. Ungureanu and C. Kruus. Bimodal content defined chunking for backup streams, In *Proc. 8th USENIX Conference on File and Storage Technologies*, 2010.

[33] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *Proc. 6th USENIX Conference on File and Storage Technologies*, 2008, pp. 1-14.

Tradeoffs in Scalable Data Routing for Deduplication Clusters

Wei Dong[*]
Princeton University

Fred Douglis
EMC

Kai Li
Princeton University and EMC

Hugo Patterson
EMC

Sazzala Reddy
EMC

Philip Shilane
EMC

Abstract

As data have been growing rapidly in data centers, deduplication storage systems continuously face challenges in providing the corresponding throughputs and capacities necessary to move backup data within backup and recovery window times. One approach is to build a cluster deduplication storage system with multiple deduplication storage system nodes. The goal is to achieve scalable throughput and capacity using extremely high-throughput (e.g. 1.5 GB/s) nodes, with a minimal loss of compression ratio. The key technical issue is to route data intelligently at an appropriate granularity.

We present a cluster-based deduplication system that can deduplicate with high throughput, support deduplication ratios comparable to that of a single system, and maintain a low variation in the storage utilization of individual nodes. In experiments with dozens of nodes, we examine tradeoffs between *stateless* data routing approaches with low overhead and *stateful* approaches that have higher overhead but avoid imbalances that can adversely affect deduplication effectiveness for some datasets in large clusters. The stateless approach has been deployed in a two-node commercial system that achieves 3 GB/s for multi-stream deduplication throughput and currently scales to 5.6 PB of storage (assuming 20X total compression).

1 Introduction

For business reasons and regulatory requirements [14, 29], data centers are required to backup and recover their exponentially increasing amounts of data [15] to and from backup storage within relatively small windows of time; typically a small number of hours. Furthermore, many copies of the data must be retained for potentially long periods, from weeks to years. Typically, backup software aggregates files into multi-gigabyte "tar" type files for storage. To minimize the cost of storing the many backup copies of data, these files have traditionally been stored on tape.

Deduplication is a technique for effectively reducing the storage requirement of backup data, making disk-based backup feasible. Deduplication replaces identical regions of data (files or pieces of files) with references (such as a SHA-1 hash) to data already stored on disk [6, 20, 27, 36]. Several commercial storage systems exist that use some form of deduplication in combination with compression (such as Lempel-Ziv [37]) to store hundreds of terabytes up to petabytes of original (logical) data [8, 9, 16, 25]. One state-of-the-art single-node deduplication system achieves 1.5 GB/s in-line deduplication throughput while storing petabytes of backup data with a combined data reduction ratio in the range of 10X to 30X [10].

To meet increasing requirements, our goal is a backup storage system large enough to handle *multiple* primary storage systems. An attractive approach is to build a deduplication cluster storage system with individual high-throughput nodes. Such a system should achieve scalable throughput, scalable capacity, and a cluster-wide data reduction ratio close to that of a single very large deduplication system. Clustering storage systems [5, 21, 30] are a well-known technique to increase capacity, but adding deduplication nodes to such clusters suffer from two problems. First, it will fail to achieve high deduplication because such systems do not route based on data content. Second, tightly-coupled cluster file systems often do not exhibit linear performance scalability because of requirements for metadata synchronization or fine-granularity data sharing.

Specialized deduplication clusters lend themselves to a loosely-coupled architecture because consistent use of content-aware data routing can leverage the sophisticated single-node caching mechanisms and data layouts [36] to achieve scalable throughput and capacity while maximizing data reduction. However, there is a tension between deduplication effectiveness and

[*] Work done in part as an intern with Data Domain, now part of EMC.

throughput. On one hand, as chunk size decreases, deduplication rate increases, and single-node systems may deduplicate chunks as small as 4-8 KB[1] to achieve very high deduplication. On the other hand, with larger chunk sizes, high throughput is achieved because of stream and inter-file locality, and per-chunk memory overhead is minimized [18, 35]. High throughput deduplication with small chunk sizes is achieved on individual nodes using techniques that take advantage of cache locality to reduce I/O bottlenecks [20, 36]. For existing deduplication clusters like HYDRAstor [8], though, relatively large chunk sizes (∼64 KB) are used to maintain high throughput and fault tolerance at the cost of deduplication. We would like to achieve scalable throughput and capacity with cluster-wide deduplication close to that of a state-of-the-art single node.

In this paper, we propose a deduplicating cluster that addresses these issues by intelligently "striping" large files across a cluster: we create *super-chunks* that represent consecutive smaller chunks of data, *route* super-chunks to nodes, and then perform deduplication at each node. We define data routing as the assignment of super-chunks to nodes. By routing data at the granularity of super-chunks rather than individual chunks, we maintain cache locality, reduce system overheads by batch processing, and exploit the deduplication characteristics of smaller chunks at each node. The challenges with routing at the super-chunk level are, first, the risk of creating duplicates, since the fingerprint index is maintained independently on each node; and second, the need for scalable performance, since the system can overload a single node by routing too much data to it.

We present two techniques to solve the data routing problem in building an efficient deduplication cluster, and we evaluate them through trace-driven simulation of collected backups up to 50 TB. First, we describe a *stateless* technique that routes based on only 64 bytes from the super-chunk. It is remarkably effective on typical backup datasets, usually with only a ∼10% decrease in deduplication for small clusters compared to a single node; for balanced workloads the gap is within ∼10-20% even for clusters of 32–64 nodes. Second, we compare the stateless approach to a *stateful* technique that uses information about where previous chunks were routed. This achieves deduplication nearly as high as a single node and distributes data evenly among dozens of nodes, but it requires significant computation and either greater memory or communication overheads. We also explore a range of techniques for routing super-chunks that trade off memory and communication requirements, including varying how super-chunks are formed, how large they are on average, how they are assigned to nodes, and how node imbalance is addressed.

The rest of this paper is organized as follows. Section 2 describes our system architecture, then Section 3 focuses on alternatives for super-chunk creation and routing. Section 4 presents our experimental methodology, datasets, and simulator, and Section 5 shows the corresponding results. We briefly describe our product in Section 6. We discuss related work in Section 7, and conclusions and future work are presented in Section 8.

2 System Overview

This section presents our deduplication cluster design. We first review the architecture of our earlier storage system [36], which we use as a single-node building block. Because the design of the single-node system emphasizes **high throughput**, any cluster architecture must be designed to support scalable performance. We then show the design of the deduplication cluster with stateless routing, corresponding to our product (differences pertaining to *stateful* routing are presented later in the paper).

We use the following criteria to govern our design decisions for the system architecture and choosing a routing strategy:

- **Throughput** Our cluster should scale throughput with the number of nodes by maximizing parallel usage of high-throughput storage nodes. This implies that our architecture must optimize for cache locality, even with some penalty with respect to deduplication capacity—we will write duplicates across nodes for improved performance, within reason.
- **Capacity** To maximize capacity, repeated patterns of data should be forwarded to storage nodes in a consistent fashion. Importantly, capacity usage should be balanced across nodes, because if a node fills up, the system must place new data on alternate nodes. Repeating the same data on multiple nodes leads to poor deduplication.

The architecture of our single-node deduplication system is shown in Figure 1(a). We assume the incoming data streams have been divided into chunks with a content-based chunking algorithm [4, 22], and a fingerprint has been computed to uniquely identify each chunk. The main task of the system is to quickly determine whether each incoming chunk is new to the system and then to efficiently store new chunks. High-throughput fingerprint lookup is achieved by exploiting the *deduplication locality* of backup datasets: in the same backup stream, chunks following a duplicate chunk are likely to be duplicates, too.

To preserve locality, we use a technique based on Stream Informed Segment[2] Layout [36]: disk storage is

[1] Throughout the paper, references to chunks of a given size refer to chunks that are expected to *average* that size.

[2] Note that the term "segment" in the earlier paper means the same

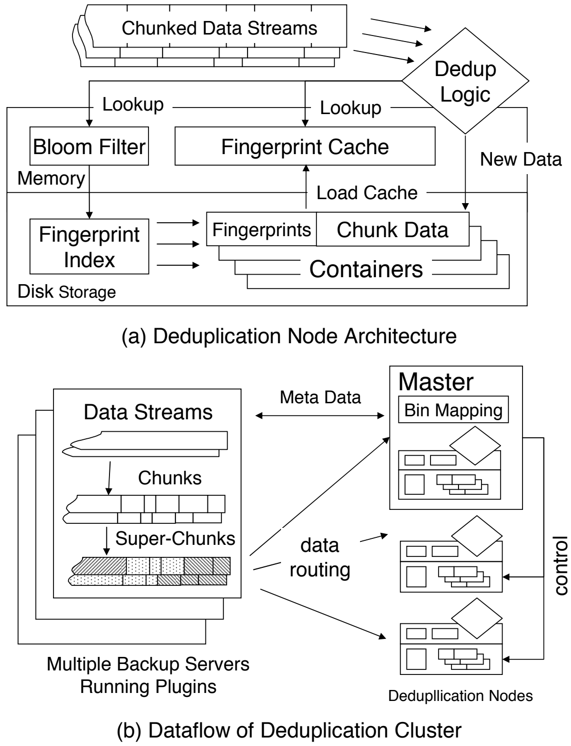


Figure 1: Deduplication node architecture and cluster design using individual nodes as building blocks.

divided into fixed-size large pieces called containers, and each stream has a dedicated container. The non-duplicate fingerprints and chunk data are appended to the metadata part and the data part of the container. The sequence of fingerprints needed to reconstruct a file is also written as chunks and stored to containers, and a root fingerprint is maintained in a directory structure. When the current container is full, it is flushed to disk, and a new container is allocated for the stream.

To identify existing chunks, a fingerprint cache avoids a substantial fraction of index lookups, and for those not found in the cache, a Bloom filter [3] identifies with high probability which fingerprints will be found in the on-disk index. Thus disk accesses only occur either when a duplicate chunk misses in our cache or when a full container of new chunks is flushed to disk. (In rare cases, a false positive from the Bloom filter will cause an unnecessary lookup to the on-disk index.) Once a fingerprint is loaded, many fingerprints that were written at the same time are loaded with it, enabling subsequent duplicate chunks to hit in the fingerprint cache.

Figure 1(b) demonstrates how to combine multiple deduplication nodes into a cluster. Backup software on each client collects individual files into a backup stream, which it transfers to a backup server. We offer a plugin [12] that runs on a customer's backup servers, which divides each stream into chunks, fingerprints them, groups them into a super-chunk, and routes each super-chunk to a deduplicating storage node. Each storage node locally applies deduplication logic to chunks while preserving data locality, which is essential to maintain high throughput.

To clarify the parallelization that takes place in our cluster, consider writing a file to the cluster. When a super-chunk is routed to a storage node, deduplication begins while the next super-chunk is created and routed to a potentially different node. All of the metadata needed to reconstruct a file is stored in chunks and distributed across the nodes. When reading back a file, parallel reads are initiated to all of the nodes by looking ahead through the metadata references and issuing reads for super-chunks to the appropriate nodes. To achieve maximum parallelization, the I/O load should be equal on each node, and both read and write throughput should scale linearly with the number of nodes.

Note that we do not yet specifically address the inter-node dependencies that arise in the event of a failure. Each node is highly redundant, with RAID and other data integrity mechanisms. It would be possible to provide redundant controllers in each node to eliminate that single point of failure, but these details are beyond the scope of this paper.

Storage Rebalancing: When super-chunks are routed to a storage node, we use a level of indirection called a *bin*. We assign a super-chunk to a bin using the mod function, and then map each bin to a given node. By using many more bins ($\sim$1000) than actual nodes, the Bin Manager (running on the master node) can rebalance nodes by reassigning bins in the future. The Bin Manager also handles expansion cases such as when a node's storage expands or when a new node is added to the cluster. In those cases, the Bin Manager reassigns bins to the new storage to maintain balanced usage. Rebalancing data takes place online while backups and other operations continue, and the entire process is transparent to the user. After a rebalance operation, the cluster will generally remain balanced for future backups. The master node communicates the bin-to-node mapping to the plugin.

Bin migration occurs when the storage usage of a node exceeds the average usage in the cluster by some threshold (defaulting to 5%). Note that if there is a great deal of skew in the total physical storage of a single bin, that bin can exceed the threshold even if it is the only bin stored on a node. Such anomalous behavior is rare but possible, and we discuss some examples of this in Section 5.

as the term "chunk" in this paper.

3 Data Routing

This section addresses two issues with data routing in our deduplication cluster: how to group chunks into super-chunks, and how to route data. Super-chunk formation is relatively straightforward and is discussed in Section 3.1. We focus here on two routing strategies: stateless routing, light-weight and well suited for most balanced workloads (Section 3.2); and stateful routing, requiring more overhead but maintaining a higher deduplication rate with larger clusters (Section 3.3).

3.1 Super-Chunk Formation

There are two important criteria for grouping consecutive chunks into super-chunks. First, we want an average super-chunk size that supports high throughput. Second, we want super-chunk selection to be resistant to small changes between full backups.

The size of a super-chunk could vary from a single chunk to many megabytes, or it could be equal to individual files as suggested by Extreme Binning [2]. We experimented with a variety of average super-chunk sizes from 8 KB up to 4 MB on backup datasets. The average super-chunk size affects deduplication, balance across storage nodes, and throughput, and it is more thoroughly explored in Section 5.3. We generally found that a 1 MB average super-chunk size is a good choice, because it results in efficient data locality on storage nodes as well as generally high deduplication, and this is the default value used in our experiments unless otherwise noted.

Determining super-chunk boundaries (anchoring) mirrors the problem of anchoring chunks [24] in many ways and should be implemented in a content-dependent fashion. Since all chunks in a super-chunk are routed together, deduplication is affected by super-chunk boundaries. We represent each chunk with a feature (see the next subsection), compare the feature against a mask, and when the mask is matched, the selected chunk becomes the boundary between super-chunks. Minimum and maximum super-chunk sizes are enforced, half and double the desired super-chunk size respectively.

3.2 Stateless Routing

Numerous data routing techniques are possible: routing based only on the contents of the current super-chunk is *stateless*, while routing super-chunks using information about the location of existing chunks is *stateful* (see Section 3.3).

For stateless routing, the basic technique is to produce a feature value representing the data and then apply a simple function (such as `mod` #bins) to the value to make the assignment. As a super-chunk is a sequence of chunks, we first compute a feature from each chunk, and then select one of those features to represent the super-chunk.

There are many options for generating a chunk feature. A hash could be calculated over an entire chunk (`hash(*)`) or over a prefix of the bytes near an anchor point (`hash(N)`, for a prefix of N bytes). Using the hash of a representative portion of a chunk results in data that are similar, but not identical, being routed to the same node; the net effect is to improve deduplication while increasing skew. We tried a range of prefix lengths and found the best results when using the first 64 bytes after a chunk anchor point (*i.e.*, `hash(64)`), which we compare to `hash(*)`. When using a hash for routing rather than deduplication, collisions are acceptable, so we use the first 32-bit word of the SHA-1 hash for `hash(64)`.

In addition, we considered other variants, such as fingerprints computed over sliding windows of content [22]; these did not make a substantial difference in the outcome, and we do not discuss them further.

To select a super-chunk feature based on the chunk features, the first, maximum, minimum, or most common chunk feature could be selected; using just the first has the advantage that it is not necessary to buffer an entire super-chunk before deciding where to route it, something that matters when hundreds or thousands of streams are being processed simultaneously. Another stateless technique is to treat the feature of each chunk as a "vote" for a node and select the most common, which does not work especially well, because hash values are often uniformly distributed. We experimented with a variety of options and found the most interesting results with four combinations: `hash(64)` of the first chunk, the minimum `hash(64)` across a super-chunk, `hash(*)` of the first chunk, and the minimum `hash(*)` across a super-chunk (compared in detail in Section 5.2). Elsewhere, `hash(64)` refers to the feature from the first chunk unless stated otherwise.

The main advantages of stateless techniques are (1) reduced overhead for recording node assignments, and (2) reduced requirements for recovering this state after a system failure. Stateless routing has some properties of a "shared nothing" [31] architecture because of limited shared state. There is a potential for a loss of deduplication compared to the single-node case, and there is also the potential for increased data skew if the selected features are not uniformly distributed. We find empirically that the reduction in deduplication effectiveness is within acceptable bounds, and bin migration can usually address excessive data skew.

3.3 Stateful routing

Using information about the location of existing chunks can improve deduplication, at an increased cost in (a) computation and (b) memory or communication. We present a stateful approach that produces deduplication that is frequently comparable to that of a single node even with a significant number of nodes (32-64); also, by balancing the benefit of matching existing chunks against the capacity of overloaded nodes, it avoids the need to migrate data after the fact. This approach is not a panacea, however, as it increases memory requirements (per-node Bloom filters, if storing them on a master node, and buffering an entire super-chunk before routing it) and computational overhead, as discussed below.

To summarize our stateful routing algorithm, in its simplest form:

1. Use a Bloom filter to count the number of times each fingerprint in a super-chunk is already stored on a given node.

2. Weight the number of matches ("votes") by each node's relative storage utilization. Overweight nodes are excluded.

3. If the highest weighted vote is above a threshold, select that node.

4. If no node has sufficient weighted votes, route to the node selected via `hash(64)` of the first chunk if it is not overloaded; otherwise route to the least loaded node.

We now explain the algorithm in more detail. To route a super-chunk, once the master node knows the number of chunks in common with (a.k.a. "matching") each node, it selects a destination. However, such a "voting" approach requires care to avoid problematic cases: simply targeting the node with the most matching chunks will route more and more super-chunks there, because the more data it has relative to other nodes, the more likely it is to match the most chunks.

Thus, one refinement to this stateful approach is to create a threshold for a minimum fraction of chunks that must match a node before it is selected. With a uniform distribution, one expects each node to match at most $\frac{C}{N}$ chunks on average, where C is the number of chunks in the super-chunk and N is the number of nodes. Typically not all chunks will match any node, and the average number of matches will be lower, but if a node already stores significantly more than the expected average, this is a reason to route the super-chunk to that node. In our system, a *voting benefit threshold* of 1.5 means that a node is considered as a candidate only if it already matches at least $\frac{1.5C}{N}$ chunks. This prevents a node from being selected simply because it matches more than any other node, when no node matches well enough to be of interest.

Simply using a static threshold for the number of matches to vote a super-chunk to a particular node still results in high data skew, as popular nodes get more popular over time. A technique we call **weighted voting** addresses that deficiency by striking a balance between deduplication and uniform storage utilization. It decreases the perceived value of known duplicates in proportion to the extent to which a node is overloaded relative to the average storage utilization of the system. As an example, if a node matches $\frac{2C}{N}$ chunks in a super-chunk, but that node stores 120% ($\frac{6}{5}$) of the average node, then the node is treated as though it matched $\frac{5}{6} * \frac{2C}{N}$ chunks. Note that while a node that stores less than the average could be given a weight < 1, increasing the overall weighted value, instead we assign such nodes a weight of 1. This ensures that when multiple nodes can easily accommodate the new super-chunk, the node is selected based on the best match. We experimented with various weight functions, but we found that it is effective simply to exclude nodes that are above a capacity threshold. In practice, a capacity of 5% above the average was selected as the threshold (see Sec 5.4).

The computational cost arises because the stateful approach computes where every chunk in a super-chunk is currently stored. A Bloom filter lookup has to be performed for each chunk, on each node in the cluster, before a routing destination can be picked. Each such lookup is extremely fast ($\sim 100-200$ns), but there can be a great many of these lookups: inserting M chunks into an N-node cluster would result in NM Bloom filter lookups, compared to M lookups in a single-node system. The additional overhead in memory or communication depends on whether the master node(s) tracks the state of each storage node (resulting in substantial memory allocations) or sends the chunk fingerprints to the storage nodes and collects counts of how many chunks match each node (resulting in communication overhead). One way to mitigate the effect is to *sample* [20] chunks that are used for voting. We reduce the number of chunks considered by checking each chunk's fingerprint for a bit pattern of a specific length (e.g., B bits must match a pattern for a $1/2^B$ sampling rate); the total number of lookups is then approximately $NM/2^B$. Without sampling, the total cost of the Bloom filter lookups is about 1.2 hours of computation for a 5-TB dataset, but a sampling rate of $1/8$ cuts this to 13 minutes of overhead with a nominal reduction in deduplication (see Section 5.5). That work can further be parallelized across back-ends or in threads on the front-end.

As an example, the general approach to weighted voting is depicted in Figure 2. In this example, the seven numbered chunks in this super-chunk are sampled for voting. Chunks 1, 3, and 4 are contained on node 1, chunks 2, 3, 5, and 6 are on node 2, chunk 5 is also on node 4, and chunk 7 is not stored on any node. Node 1 has 3 raw votes, and node 2 has 4. Factoring in space, since node 2 uses much more than the average,

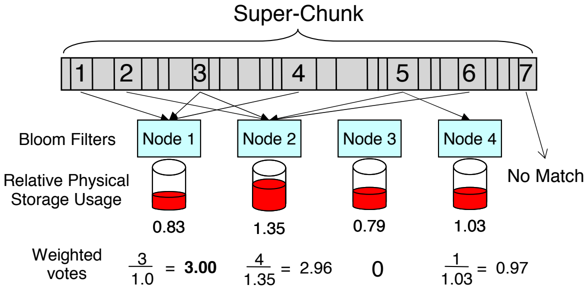


Figure 2: Weighted voting example. A node with many matches will be selected if it does not also have too much data already, relative to the other nodes. Any node with a relative storage usage of less than 1 is treated as though it is at the average.

its weighted votes are $(4/1.35) = 2.96$. Node 1 has a slightly higher weighted vote of 3. The minimum weight for a node to be selected is $\frac{1.5\times7}{4} = 2.6$. Thus node 1 is selected for routing.

The main advantage of a stateful technique is the opportunity to incorporate expected deduplication and capacity balancing while assigning chunks to nodes. On the other hand, computational or communication overhead must be considered when choosing this technique, though it is an attractive option for coping with unbalanced workloads or cluster sizes beyond our current expectations.

4 Experimental Methodology

We use trace-driven simulation to evaluate the tradeoffs of the various techniques described in the previous section. This section describes the datasets used, the evaluation metrics, and the details of the simulator.

4.1 Datasets

In this paper, we simulate super-chunk routing for nine datasets. Three were collected from large backup environments representing typical scenarios where a backup server hosts multiple data types from dozens of clients. These datasets contain approximately 40-50 TB precompressed data. To analyze how our routing technique handles datasets with specific properties, we also analyze five datasets representing single data types. Four of the datasets are each approximately 5 TB and a fifth is about 13 TB. In addition, we synthesize a "blended" dataset consisting of a mixture of the five smaller datasets. In general, we use them in the form that a deduplication appliance would see them: tar files that are usually many gigabytes in size, rather than individual small files. With the exception of the "blended" dataset, all of these datasets represent real backups from production environments.

Name	Size (GB)		Dedup.	Months
	Total	Peak		
Collection 1	40,695	2,867	6.1	1–2
Collection 2	44,536	1,536	11.5	4–6
Collection 3	51,584	2,150	6.1	3
Perforce	4,574	250	20.8	6
Workstations	4,926	200	5.6	6
Exchange	5,253	33	6.8	7
System Logs	5,436	122	38.7	4
Home Dirs.	12,907	855	19.3	3
Blended	33,097	N/A	12.5	N/A

Table 1: Summary of datasets. The Collection datasets were collected from backup servers with multiple data types, and the other datasets were collected from single data-type environments. Deduplication ratios are obtained from a single-node system.

For the three collected datasets, we received permission to analyze production backup servers within EMC. We gathered traces for each file including the timestamp, sequence of chunk fingerprints, and other metadata necessary to analyze chunk routing. At an earlier collection on internal backup servers, we gathered copies of backup files for the individual data types.

Table 1 lists salient information of these datasets: the total logical size, the daily peak size, the single-node deduplication rate, and the number of months in the 99th percentile of retention period. The datasets are:

Collection 1: Backups from approximately 100 clients consisting of half software development and half business records. Backups are retained 1-2 months.

Collection 2: Backups from approximately 50 engineering workstations with 4 months of retention and servers with 6 months of retention.

Collection 3: Backups of over 100 clients for Exchange, SQL servers, and Windows workstations with 3 months of retention.

Perforce: Backups from a version control repository.

Workstations: Backups from 16 workstations used for build and test.

Exchange: Backups from a Microsoft Exchange server. Each day contains a single full backup.

System Logs: Backups from a server's /var directory, containing numerous system files and logs. Full backups were created weekly.

Home Directory: Backups from engineers' home directories, containing source code, office documents, etc. Full backups were created weekly.

Blended: To explore the effects of multiple datasets being written to a storage system (a common scenario), we created a blended dataset. We combined alternating super-chunks of the single data-type datasets, weighted by overall size; thus there are approximately two super-chunks from the "home directory" dataset for each super-chunk of the others. The overall deduplication for this dataset (12.5) is somewhat higher than the weighted average across the datasets (12.3), due to some cross-dataset commonality.

While our experiments studied all of these datasets, because of space limitations, we typically only present results for two: `Workstations` and `Exchange`. Experiments with `Workstations` have results consistent with the other datasets and represents our expected customer experience. The `Exchange` dataset showed consistently worse results with our techniques and is presented for comparison. Because of data patterns within `Exchange`, using a 1-MB super-chunk results in overloading a single bin with $1/16$ of the data.

4.2 Evaluation Metrics

The principal evaluation metrics are:

Total Deduplication (TD): The ratio of the original dataset size to the size after identical chunks are eliminated. (We do not consider *local compression* (e.g., Lempel-Ziv [37]), which is orthogonal to the issues considered in this paper.)

Data Skew: The ratio of the largest node's physical (post-deduplication) storage usage to the average usage, used to evaluate how far from this perfect balance a particular configuration is. High skew leads to a node filling up and duplicate data being written to alternative nodes, as discussed in Section 2.

Effective Deduplication (ED): Total Deduplication divided by Data Skew, as a single utility measure that encompasses both deduplication effectiveness and storage imbalance. ED is equivalent to Total Deduplication computed as if every node consumes the same amount of physical storage as the most loaded node. This metric is meaningful because the whole cluster degrades when one node is filled up. ED permits us to compare routing techniques and parameter options with a single value.

Normalized ED: ED divided by deduplication achieved by a single-node system. This is an indication of how close a super-chunk routing method is to the ideal deduplication achievable on a cluster system. It allows us to compare the effectiveness of chunk-routing methods across different datasets under the same $[0,1]$ scale.

Fingerprint Index Lookups: Number of on-disk index lookups, used as an approximation to throughput. The lookup *rate* is the number of lookups divided by the number of chunks processed by a storage node.

4.3 Simulator

Most of the results presented in this paper come from a set of simulations, organized as follows:

1. For the Collection datasets, we read from a deduplicating storage node and reconstructed files based on metadata to create a full trace including the chunk size, its `hash(*)` value, and its `hash(64)` value. The other datasets were preprocessed by reading in each file, computing chunks of a particular average size (typically 8 KB), and storing a trace.

2. The per-chunk data are passed into a program to determine super-chunk boundaries and route those super-chunks to particular nodes. It produces statistical information about deduplication rates, data skew, the number of Bloom filter lookups performed, and so on. In addition, it logs the SHA1 hash and location of each super-chunk, on a per-node basis. Its parameters include the super-chunk routing algorithm; the average super-chunk size (typically **1 MB**); the maximum relative node size before bin migration is performed (for stateless) or node assignment is avoided (for stateful), defaulting to **1.05**; some stateful routing parameters described below, and several others not considered here.

The simulator was validated in part by comparing deduplication results for Total Deduplication and skew to the values reported by the live two-node system. Due to minor implementation differences, normalized TD is typically up to 2–3% higher in the simulator than in the live system, though in one case the real system reported slightly higher normalized deduplication. Skew is similarly close.

The stateful routing parameters are: (a) Vote sampling: what fraction of chunks, on average, should be passed to the Bloom filters and checked for matches? (Default: **1/8**.) (b) Vote threshold: how many more matches than the average (as a fraction) should an average-sized node be, before being used rather than the node routed by the first chunk? (Default: **1.5**)

3. To analyze caching effects on a storage system, each of the node-specific super-chunk files can be used to synthesize a data stream with the same deduplication patterns and chunk sizes, which speeds up experimentation relative to reading the original data repeatedly. For simplicity, the compression for the synthesized chunks was fixed at 2:1, a close approximation to overall compression for the datasets used. This stream is then written to a deduplication appliance, sending each bin to its final node in the original simulations after migration.

The accuracy of using a synthesized stream in place of the original dataset was validated by comparing Total Deduplication of several synthesized results to those of original datasets.

5 Experimental Results

We focused our experiments on analyzing the impact of super-chunk routing on capacity and fingerprint index lookups across a range of cluster sizes and a variety of datasets. We start by surveying how different routing ap-

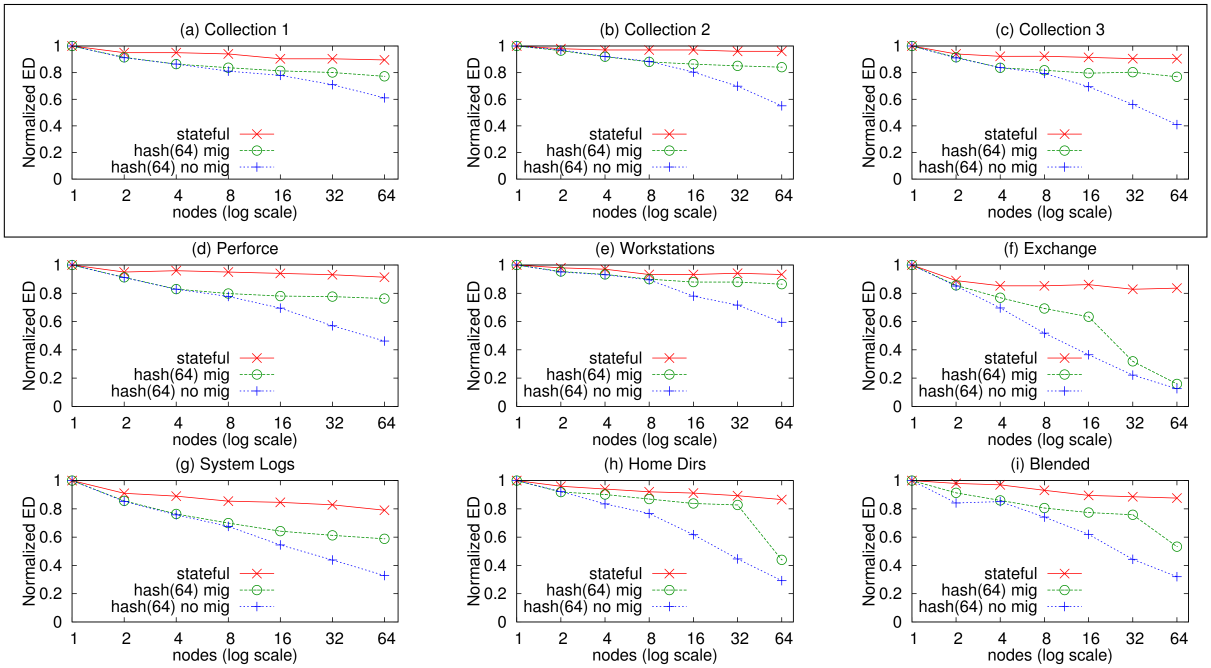


Figure 3: Normalized ED of the stateless and stateful techniques as a function of the number of nodes. The top row represents the collected "real-world" datasets. Stateful and `hash(64)` (mig) use a capacity threshold of 5%.

proaches fare over a broad range of datasets and cluster sizes (Section 5.1). This gives a picture of how Total Deduplication and skew combine into the Effective Deduplication metric. Then we dive into specifics:

- What is the best feature (`hash(64)` vs. `hash(*)`, routing by first chunk vs. all chunks in a super-chunk) for routing super-chunks (Section 5.2)?
- How does super-chunk size affect fingerprint cache lookups and locality (Section 5.3)?
- How sensitive is the system to various parameter settings, including capacity threshold (Section 5.4) and those involved in stateful routing (Section 5.5)?

5.1 Overall Effectiveness

We first compare the basic techniques, stateless and stateful, across a range of datasets. Figure 3 shows a scatter plot for the nine datasets and three algorithms: `hash(64)` without bin migration, `hash(64)` with a 5% migration threshold, and stateful routing with a 5% capacity limitation.

In general, `hash(64)` without migration works well for small clusters (2–4 nodes) but degrades steadily as the cluster size increases. Adding bin migration greatly improves the ED for most of the datasets, though even with bin migration, ED for `Exchange` decreases rapidly as the number of nodes increases, and there is also a sharp decrease for `Home Directories` and `Blended` at 64 nodes. This skew occurs when a single bin is substantially larger than the average node utilization (see Section 5.4). Stateful routing is often within 10% of the single-node deduplication even at 64 nodes, although for some datasets the gap is closer to 20%. However, there is additional overhead, as discussed in Section 5.5.

Table 2 presents normalized Total Deduplication (TD), data skew, and Effective Deduplication (ED) for several datasets, as the number of nodes varies (corresponding to the `hash(64)` (mig) and stateful curves in Figure 3). It shows how a moderate increase in skew results in a moderate reduction in ED (`Workstations`), but `Exchange` suffers from both repeated data (losing $\frac{1}{3}$ of TD) and significant skew (further reducing ED by a factor of 4).

5.2 Feature Selection

As discussed in Section 3.2, there are a number of ways to route a super-chunk. Here we compare four super-chunk features: `hash(64)` of the *first* chunk, the minimum of all `hash(64)`, the `hash(*)` of the first chunk, or the minimum of all `hash(*)`. We also compare against the method used by HYDRAstor [8], which consists of 64-KB chunks routed based on their fingerprint. Figure 4 shows the normalized ED of these four features for two datasets, not factoring in any capacity limitations. For `Workstations`, all four choices are similarly effective, which is consistent with the other datasets that are not

#	hash(64)			stateful		
nodes	TD	Skew	ED	TD	Skew	ED
Collection 1						
1	1.00	1.00	1.00	1.00	1.00	1.00
2	0.93	1.02	0.91	0.95	1.00	0.95
4	0.89	1.03	0.86	0.95	1.00	0.95
8	0.86	1.03	0.84	0.95	1.01	0.94
16	0.85	1.04	0.81	0.94	1.04	0.91
32	0.83	1.04	0.80	0.94	1.04	0.91
64	0.83	1.07	0.77	0.94	1.05	0.90
Collection 2						
1	1.00	1.00	1.00	1.00	1.00	1.00
2	0.97	1.00	0.97	0.98	1.00	0.98
4	0.94	1.02	0.92	0.97	1.00	0.97
8	0.92	1.04	0.88	0.97	1.00	0.97
16	0.90	1.04	0.86	0.97	1.00	0.97
32	0.88	1.04	0.85	0.96	1.00	0.96
64	0.87	1.04	0.84	0.96	1.00	0.96
Collection 3						
1	1.00	1.00	1.00	1.00	1.00	1.00
2	0.92	1.01	0.92	0.95	1.01	0.94
4	0.88	1.05	0.84	0.95	1.03	0.93
8	0.85	1.04	0.82	0.96	1.04	0.92
16	0.84	1.05	0.80	0.96	1.05	0.91
32	0.83	1.03	0.80	0.95	1.05	0.91
64	0.82	1.07	0.77	0.95	1.05	0.91
Workstations						
1	1.00	1.00	1.00	1.00	1.00	1.00
2	0.97	1.02	0.95	0.98	1.00	0.98
4	0.95	1.02	0.93	0.98	1.01	0.97
8	0.94	1.04	0.90	0.98	1.04	0.94
16	0.92	1.05	0.88	0.98	1.04	0.94
32	0.91	1.04	0.88	0.97	1.03	0.94
64	0.91	1.05	0.86	0.97	1.04	0.93
Exchange						
1	1.00	1.00	1.00	1.00	1.00	1.00
2	0.86	1.01	0.86	0.89	1.00	0.89
4	0.78	1.01	0.77	0.87	1.02	0.85
8	0.72	1.04	0.69	0.87	1.02	0.85
16	0.68	1.08	0.63	0.87	1.01	0.86
32	0.67	2.09	0.32	0.87	1.05	0.83
64	0.65	4.12	0.16	0.87	1.04	0.83

Table 2: Total Deduplication (TD), data skew, and normalized Effective Deduplication ratio ($ED = \frac{TD}{skew}$) for some of the datasets, using capacity thresholds of 5%.

shown. `Exchange` demonstrates the extreme case, in which most chunk-routing features degrade badly with large clusters. One can see the effect of high skew when a common feature results in distinct chunks being routed to the same node. This is less common when the entire chunk's hash is used than when a prefix is used: `first hash(*)` spreads out the data more, resulting in less data skew and better ED. Even though chunks are consistently routed with the HYDRAstor technique (`HYDRAstor`), the ED is generally worse than the other techniques because of the larger chunk size: the deduplication is less than half that achieved with 8-KB chunks on a single node.

The figure demonstrates that `first hash(64)` is generally somewhat better for smaller clusters, while `first hash(*)` is better for larger ones. (This effect arises because `first hash(64)` is more likely to keep putting even somewhat similar chunks on the same node, which improves deduplication but increases skew.) Using the minimum of either feature, as Extreme Binning does for `hash(*)`, generally achieves similar deduplication to using the first chunk. Due to its effectiveness with the cluster sizes being deployed in the near future and its reduction in buffer requirements, we use `first hash(64)` as the default and refer to it as `hash(64)` for simplicity elsewhere.

5.3 Factors Impacting Cluster Throughput

A major goal of our architecture is to maximize throughput as the cluster scales, and in a deduplicating system, the main throughput bottleneck is fingerprint index lookups that require a random disk read [36]. We are not able to produce a throughput measure in MB/s through simulation, so we use fingerprint index lookups as an indirect measure of throughput.

There are two important issues involving fingerprint index lookups to consider. The first is the total number of fingerprint index lookups that take place, since this is a measure of the amount of work required to process a dataset and is impacted by data skew. The second is the rate of fingerprint index lookup, which indicates the locality of data written to disk. These values are impacted both by the super-chunk size and number of nodes in a cluster, and we have selected a relatively large cluster size (32 nodes) while varying the super-chunk size.

Early generations of backups (the first few weeks of a dataset) tend to be laid out sequentially because of a low deduplication rate, while higher generations of backups are more scattered. To highlight this impact, we analyzed the caching effects while writing the final 1 TB of each synthesized dataset across the *N* nodes. In these experiments, the cache size is held at 12,500 fingerprints. While this may seem small, it is similar to a cache of 400,000 fingerprints on a single, large node, Also, a cache must handle multiple backup streams, while our experiments use one dataset at a time.

Figure 5 shows the skew of the uncompressed (logical) data, maximum normalized total number of fingerprint index lookups, maximum normalized fingerprint index lookup rate, and ED when routing super-chunks of various sizes for (a) `Workstations` and (b) `Exchange`. Note that we report skew of the logical data here instead of skew of the post-dedupe data reported elsewhere, because fingerprint lookups happen on logical data. The

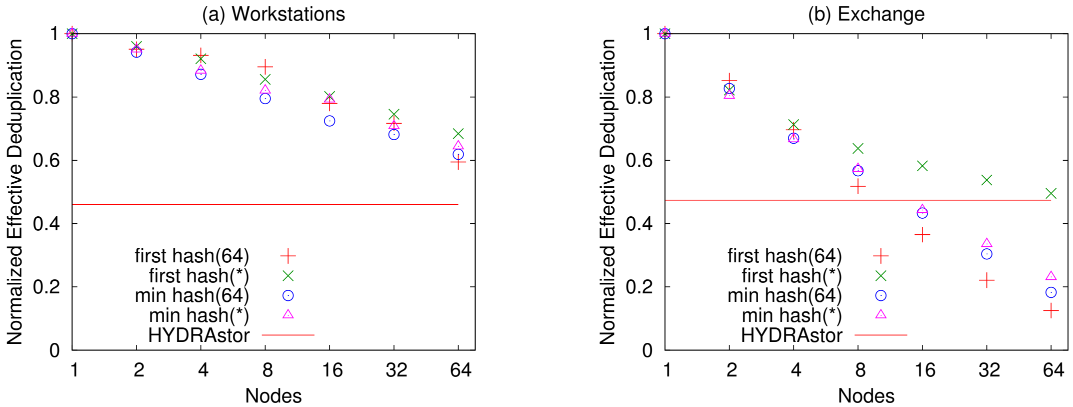


Figure 4: Normalized ED versus number of nodes with various features. No bin migration is performed. The HYDRAstor points represent 64-KB chunks routed without super-chunks, with virtually no data skew but significantly worse deduplication in most cases. *Workstations is representative of many other datasets, while Exchange is anomalous.*

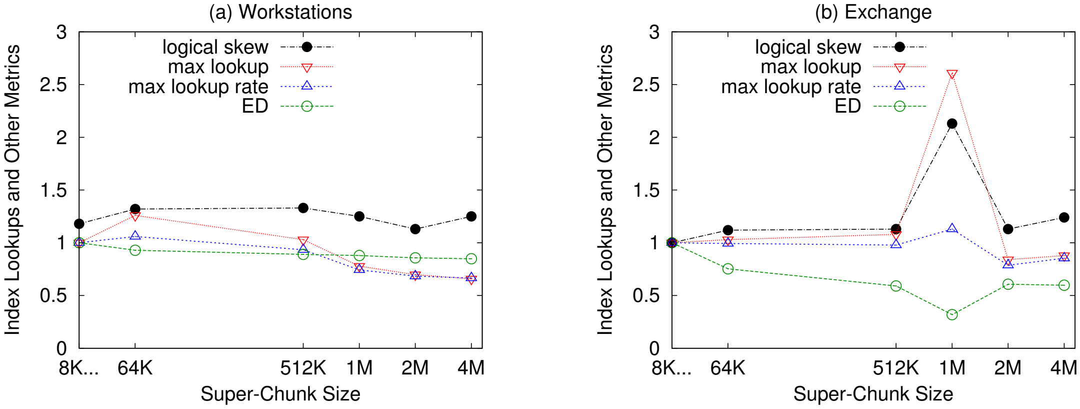


Figure 5: Skew of data written to nodes (pre-deduplication), maximum number of fingerprint index lookups and lookup rate, and ED versus super-chunk size for a 32-node cluster. Fingerprint index lookup values are normalized relative to those metrics when routing individual 8-KB chunks. As the super-chunk size increases, the maximum number of on-disk index lookups decreases for Workstations (improving throughput), while effective deduplication decreases. *Workstations is representative of many other datasets, while Exchange is anomalous.*

fingerprint index lookup numbers are normalized relative to the rate seen when routing individual 8-KB chunks. Because the lookup rate improvement achieved by using larger super-chunk sizes generally comes with a cost of lower deduplication, we also plot normalized ED to aid the selection of an appropriate super-chunk size. It should be noted that we found smaller differences in lookup rate and total number of lookups with smaller clusters.

For Workstations, we see that the total number of fingerprint index lookups and rate generally shrink as we use larger super-chunk sizes. Routing 4-MB super-chunks results in $\sim 65\%$ of the maximum total index lookups compared to routing chunks. Though data skew, maximum lookup rate, and maximum number of lookups tend to follow the same trends, the values for maximum number of lookups and maximum lookup rate may come from different nodes.

The index lookups (both total and rate) for Exchange around 1 MB highlights a case where our technique may perform poorly due to a frequently repeating pattern in the data set that causes a large fraction of the hash(64) values to map to the same bin. With smaller super-chunk sizes, less data are carried with each super-chunk, so skew can be reasonably balanced via migration, and for larger super-chunks, the problematic hash(64) value is no longer selected. For this dataset, a super-chunk size of 1 MB results in higher skew that lowers ED, and it has a high total number of lookups and worst-case cache miss rate. This is a particularly difficult example for our sys-

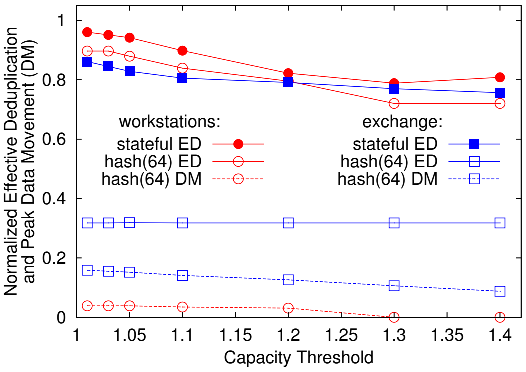


Figure 6: Normalized ED as a function of capacity threshold on 32 nodes, for `hash(64)` and stateful, and peak fraction of data movement (DM) for `hash(64)`. Note that lower points are better for data movement, while higher is better for ED.

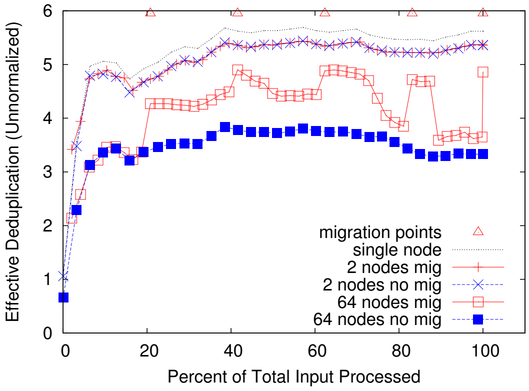


Figure 7: Effective Deduplication as a function of the amount of data processed, with and without bin migration at a 5% threshold, for the `Workstations` dataset on 2 and 64 nodes. Migration points are marked along the top, every 1 TB, with deduplication computed every 0.1 TB. The deduplication for a single node is depicted as the top curve.

tem as the same node had both the highest lookup rate and skew, which roughly multiply together to equal total lookups.

Although any particular super-chunk size can potentially result in skew if patterns in the data result in one bin being selected too often, the problem is rare in practice. Thus, despite this one poor example, we decided that 1-MB super-chunks provide both reasonable throughput and deduplication and use that as the default super-chunk size in our other experiments.

The scalability of our cluster design could more thoroughly be analyzed with a comparison of the number of fingerprint index lookups for various cluster sizes relative to the single node case. Intuitively, a single-node system might have similar lookup characteristics to nodes in a cluster when routing very large super-chunks and without data skew.

5.4 Space Usage Thresholds

Limitations on storage use arise in two contexts. For stateless routing, we periodically migrate bins away from nodes storing more than the average, if they exceed a fixed threshold relative to the mean. In the simulations, bin migration takes place after multiple 1-TB *epochs* have been processed, totaling $\sim 20\%$ of a given dataset. This means that we attempt migrations approximately 5 times per dataset regardless of size, plus once more at the end, if needed. For stateful routing, we refrain from placing new data on a node that is already storing more than that threshold above the average.

Figure 6 demonstrates the impact of the capacity threshold on ED and peak data movement, using the `Workstations` and `Exchange` datasets on 32-nodes. The top four curves show ED: for `Workstations`, deduplication effectiveness improves with increasingly tight capacity bounds, although the benefit below 5% is minimal, while for `Exchange`, the existence of a single oversized bin when using 1 MB super-chunks ensures a large skew regardless of threshold in the case of `hash(64)`.

The bottom two curves provide an indication of the impact of bin migration on data movement, as the threshold changes. We compute the fraction of data moved from a node at the end of an epoch, relative to the amount of physical data stored on the node at the time of the migration, and report the maximum across all nodes and epochs. `Exchange` moves 15–20% of the incoming data (which is on the order of $\frac{1}{32}$ of 1 TB) without improving ED, while we would migrate at most a few percent of one node's data for `Workstations`. Note that across the entire dataset, migration accounts for at most $\frac{1}{1000}$ of the data, and on the 2-node commercial systems currently deployed, they have never occurred. Because at 32 nodes we do see small amounts of migration even for the `Workstations` dataset, and increasing the threshold from 1.01 to 1.05 reduces the total data migrated by nearly a factor of 2 without much of an impact on ED, we use 1.05 as the default threshold in other experiments.

Figure 7 shows the impact of bin migration over time on the `Workstations` dataset. The curves for 2 nodes are identical, as no migration was performed. The curves for 64 nodes are significantly different, with the curve without migration having much worse ED. However, even with migration, the ED drops between migration points due to increasing skew. Note that this graph does not normalize deduplication relative to a single node, in order to highlight the effect of starting with entirely new

Sampling Rate (1-in-N)	**Workstations**		**Exchange**		**Memory (GB)**
	ED	**Lookups (B)**	**ED**	**Lookups (B)**	
1	5.28	20.57	5.74	21.95	96
2	5.27	10.83	5.72	11.62	48
4	5.27	6.03	5.76	6.46	24
8	5.31	3.61	5.63	3.88	12
16	5.27	2.41	5.46	2.59	6
32	5.19	1.81	5.13	1.95	3

Table 3: The ED, Bloom filter lookups in billions, and Bloom filter memory requirements in a 32-node system, for two of the datasets. They vary as a function of the sampling rate: which chunks are checked for existence on each node. The memory requirement is independent of the dataset.

data, then increasing deduplication over time.

5.5 Parameters for Stateful Routing

In addition to capacity limitations, stateful routing is parameterized by vote sampling and vote threshold as explained in Section 4.3. Sampling has a great impact on the number of fingerprint lookups, while surprisingly, the system is not very sensitive to a threshold requiring a node to be a particularly good match to be selected.

We evaluated sampling across a variety of datasets and cluster sizes, varying the selectivity of the anchors used to vote from 1 down to $\frac{1}{32}$. Table 3 reports the effect of sampling on ED and Bloom filter lookups for two of the datasets. (Slight rises in ED with less frequent sampling result from slightly *lower* skew due to not matching a node quite as often.) The last column of the table shows the size of a Bloom filter *on a master node* for a 1% false positive rate and up to 20 TB of unique 8-KB chunks on each node; it demonstrates how the aggregate memory requirement on the master would decrease as the sampling rate decreases. The required size to track each node is multiplied by the number of nodes, 32 in this experiment. Each node would also have its own local Bloom filter, which would be unaffected by the sampling rate used for stateful routing. If lookups are forwarded to each node, sampling would be used to limit the number of lookups, but the per-node Bloom filters used for deduplication would be used for routing, and no extra memory would be required.

We found that the ED is fairly constant from looking up all chunks (a sampling rate of 1) down to a rate of $\frac{1}{8}$ and often similar when sampling $\frac{1}{16}$; it degrades significantly, as expected, when less than that. Thus we use a default of $\frac{1}{8}$ for stateful routing elsewhere in this paper.

We also examined the vote benefit threshold. While we use a default of 1.5, the system is not very sensitive to values from 0.75 to 2. The key is to have a high enough threshold that a single chunk will not "attract" more and more dissimilar chunks due to one match.

6 Cluster Deduplication Product

EMC now makes a product based on this technology [13]. The cluster configuration currently consists of two nodes and uses the `hash(64)` routing technique with bin migration. Each node has the following hardware configuration: 4 socket processor, 4 cores per socket, and each core is running at 2.93 Ghz; 64 GB of memory; four 10-Gb Ethernet interfaces (one for external traffic and one for inter-node traffic, both in a fail-over pair); and 140 TB of storage, consisting of 12 shelves of 1-TB drives. Each shelf has 16 drives in a 12+2 RAID-6 configuration with 2 spare drives.

The total physical capacity of the two-node system is 280 TB. Under typical backup usage, total compression is expected to be 20X, which leads to a logical capacity of 5.6 PB of storage. Write performance with multiple streams is over 3 GB/s. Note that this performance was achieved with processing on the backup server as described in Section 2, which communicates with storage nodes to filter duplicate chunks before network transfer. Because of the filtering step, logical throughput (file size divided by transfer time) can even exceed LAN speed.

We measured the steady-state write and read performance with 1–4 nodes and found close to linear improvement as the number of nodes increases. While simulations suggest our architecture will scale to a larger number of nodes, we have not yet tuned our product for a larger system or run performance tests.

In over six months of customer usage, bin migration has never run, which indicates stateless routing typically maintains balance across two nodes.

7 Related Work

Chunk-based deduplication is the most widely used deduplication method for secondary storage. Such a system breaks a data file or stream into contiguous chunks and eliminates duplicate copies by recording references to previous, identical chunks. Numerous studies have investigated content-addressable storage using whole files [1], fixed-size blocks [27, 28], content-defined chunks [17, 24, 36], and combinations or comparisons of these approaches [19, 23, 26, 32]; generally, these have found that using content-defined chunks improves deduplication rates when small file modifications are stored. Once the data are divided into chunks, it is represented by a secure fingerprint (*e.g.*, SHA-1) used for deduplication.

A technique to decrease the in-memory index requirements is presented in Sparse Indexing [20], which uses a sampling technique to reduce the size of the fingerprint index. The backup set is broken into relatively large regions in a content-defined manner similar to our super-chunks, each containing thousands of chunks. Regions are then deduplicated against a few of the most similar regions that have been previously stored using a sparse, in-memory index with only a small loss of deduplication.

While Sparse Indexing is used in a single system to reduce its memory footprint, the notion of sampling within a region of chunks to identify other chunks against which new data may be deduplicated is similar to our sampling approach in stateful routing. However, we use those matches to direct to a specific node, while they use matches to load a cache for deduplication.

Several other deduplication clusters have been presented in the literature. Bhagwat *et al.* [2] describe a distributed deduplication system based on "Extreme Binning": data are forwarded and stored on a file basis, and the representative chunk ID (the minimum of all chunk fingerprints of a file) is used to determine the destination. An incoming file is only deduplicated against a file with a matching representative chunk ID rather than against all data in the system. Note that Extreme Binning is intended for operations on individual files, not aggregates of all files being backed up together. In the latter case, this approach limits deduplication when inter-file locality is poor, suffers from increased cache misses and data skew, and requires multiple passes over the data when these aggregates are too big to fit in memory.

DEBAR [34] also deduplicates individual files written to their cluster. Unlike our system, DEBAR deduplicates files partially as they are written to disk and completes deduplication during post-processing by sharing fingerprints between nodes.

HYDRAstor [8] is a cluster deduplication storage system that creates chunks from a backup stream and routes chunks to storage nodes, and HydraFS [33] is a file system built on top of the underlying HYDRAstor architecture. Throughput of hundreds of MB/s is achieved on 4-12 storage nodes while using 64 KB-sized chunks. Individual chunks are routed by evenly partitioning fingerprint space across storage nodes, which is similar to the routing techniques used by Avamar [11] and PureDisk [7]. In comparison, our system uses larger super-chunks for routing to maximize cache locality and throughput but also uses smaller chunks for deduplication to achieve higher deduplication.

Choosing the right chunking granularity presents a tradeoff between deduplication and system capacity and throughput even in a single-node system [35]. Bimodal chunking [18] is based on the observation that using large chunks reduces metadata overhead and improves throughput, but large chunks fail to recover some deduplication opportunities when they straddle the point where new data are added to the stream. Bimodal chunking tries to identify such points and uses a smaller chunk size around them for better deduplication.

8 Conclusion and Future Work

This paper presents super-chunk routing as an important technique for building deduplication clusters to achieve scalable throughput and capacity while maximizing effective deduplication. We have investigated properties of both stateless and stateful versions of super-chunk routing. We also describe a two-node deduplication storage product that implements the stateless method to achieve 3 GB/sec deduplication throughput with the capacity to store approximately 5.6 PB of backup data.

Our study has three conclusions. First, we have found that using super-chunks, a multiple of fine-grained deduplication chunks, for data routing is superior to using individual chunks to achieve scalable throughput while maximizing deduplication. We have demonstrated that a 1-MB super-chunk size is a good tradeoff between index lookups, which directly impact deduplication throughput, and effective cluster-wide deduplication.

Second, the stateless routing method (`hash(64)`) with bin migration is a simple and yet efficient way to build a deduplication cluster. Our simulation results on real-world datasets show that this method can achieve good balance and scalable throughput (good caching locality) while achieving at least 80% of the single-node effective deduplication, and bin migration appears to be critical to the success of the stateless approach in larger clusters.

Third, our study shows that effective deduplication of the stateless routed cluster for certain datasets (most notably `Exchange`) may drop quickly as the number of nodes increases beyond 4. To solve this problem, we have proposed a stateful data routing approach. Simulations show this approach can achieve 80% or better normalized ED when using up to 64 nodes in a cluster, even for "pathological" cases.

Several issues remain open. First, we would like to further our understanding of the conditions that cause severe data skew with the stateless approach. To date, no bin migration has occurred in the production system described in this paper; this is not surprising considering that ED for `hash(64)` on two nodes is virtually identical for each of our datasets, with or without bin migration. The same is true for most, but not all, of the datasets as the cluster size increases moderately. Second, we plan to examine the scalability of the system across a broad range of cluster sizes and the impact of parameters such as feature selection and super-chunk size. Third, we want to explore the use of bin migration to support reconfiguration such as node additions. Finally, we plan to build a prototype cluster with stateful routing so that more thorough experiments can be conducted in lab and in customer environments.

Acknowledgments

We thank Dhanabal Ekambaram, Paul Jardetzky, Ed Lee, Dheer Moghe, Naveen Rastogi, Pratap Singh, and Grant Wallace for helpful comments and/or assistance with our experimentation. We are especially grateful to the anonymous referees and our shepherd, Cristian Ungureanu, for their feedback and guidance.

References

[1] 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 *OSDI '02: Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 1–14, 2002.

[2] D. Bhagwat, K. Eshghi, D. D. Long, and M. Lillibridge. Extreme binning: scalable, parallel deduplication for chunk-based file backup. In *MASCOTS 09: Proceedings of the 17th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, Sept. 2009.

[3] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[4] S. Brin, J. Davis, and H. García-Molina. Copy detection mechanisms for digital documents. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 398–409, 1995.

[5] 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 391–430. MIT Press, 2000.

[6] L. P. Cox., C. D. Murray, and B. D. Noble. Pastiche: making backup cheap and easy. In *OSDI '02: Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 285–298, New York, NY, USA, 2002. ACM.

[7] M. Dewaikar. Symantec NetBackup PureDisk: optimizing backups with deduplication for remote offices, data center and virtual machines. `http://eval.symantec.com/mktginfo/enterprise/white_papers/b-symantec_ne%tbackup_puredisk_WP-en-us.pdf`, September 2009.

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

[9] EMC Corporation. Data Domain products. `http://www.datadomain.com/products/`, 2009.

[10] EMC Corporation. DD880: deduplication storage for the core data center. `http://www.datadomain.com/pdf/DataDomain-DD880-Datasheet.pdf`, 2009.

[11] EMC Corporation. Efficient data protection with EMC Avamar global deduplication software. `http://www.emc.com/collateral/software/white-papers/h2681-efdta-prot-av%amar.pdf`, July 2009.

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

[13] EMC Corporation. Data Domain Global Deduplication Array, 2010. `http://www.datadomain.com/products/global-deduplication-array.html`.

[14] European Parliament. Directive 2006/24/EC "On the retention of data generated or processed in connection with the provision of publicly available electronic communications services or of public communication networks" , March 2006.

[15] J. F. Gantz, C. Chute, A. Manfrediz, S. Minton, D. Reinsel, W. Schlichting, and A. Toncheva. The diverse and exploding digital universe: an updated forecast of worldwide information growth through 2011. An IDC White Paper — sponsored by EMC, March 2008.

[16] IBM Corporation. IBM ProtecTIER Deduplication Solutions, 2010. `http://www-03.ibm.com/systems/storage/tape/protectier`.

[17] N. Jain, M. Dahlin, and R. Tewari. Taper: tiered approach for eliminating redundancy in replica synchronization. In *FAST '05: Proceedings of the 4th USENIX Conference on File and Storage Technologies*, pages 21–21, 2005.

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

[19] P. Kulkarni, F. Douglis, J. LaVoie, and J. M. Tracey. Redundancy elimination within large collections of

files. In *Proceedings of the USENIX Annual Technical Conference*, pages 59–72, 2004.

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

[21] The Lustre File System, 2010. `http://www.lustre.org`.

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

[23] D. T. Meyer and W. J. Bolosky. A Study of Practical Deduplication. In *FAST '11: Proceedings of the 9th Conference on File and Storage Technologies*, February 2011.

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

[25] Network Appliance. NetApp ONTAP. `http://www.netapp.com/us/products/platform-os/dedupe.html`, 2009.

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

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

[28] S. C. Rhea, R. Cox, and A. Pesterev. Fast, inexpensive content-addressed storage in foundation. In *Proceedings of the USENIX Annual Technical Conference*, pages 143–156, 2008.

[29] 107th Congress, United States of America. Public Law 107-204: "Sarbanes-Oxley Act of 2002", July 2002.

[30] S. R. Soltis, T. M. Ruwart, and M. T. Okeefe. The global file system. In *MSS '96: Proceedings of the 5th NASA Goddard Conference on Mass Storage*, pages 319–342, 1996.

[31] M. Stonebraker. The case for shared-nothing. *IEEE Database Engineering*, 9(1), March 1986.

[32] N. Tolia, M. Kozuch, M. Satyanarayanan, B. Karp, T. Bressoud, and A. Perrig. Opportunistic Use of Content Addressable Storage for Distributed File Systems. In *Proceedings of the USENIX Annual Technical Conference*, pages 127–140, 2003.

[33] C. Ungureanu, B. Atkin, A. Aranya, S. Gokhale, S. Rago, G. Calkowski, C. Dubnicki, and A. Bohra. HydraFS: a high throughput file system for the HYDRAstor content-addressable storage system. In *FAST '10: Proceedings of the 8th Conference on File and Storage Technologies*, February 2010.

[34] T. Yang, D. Feng, Z. Niu, K. Zhou, and Y. Wan. DEBAR: a scalable high-performance deduplication storage system for backup and archiving. In *IEEE International Symposium on Parallel & Distributed Processing*, May 2010.

[35] L. You and C. Karamanolis. Evaluation of efficient archival storage techniques. In *MSS '04: Proceedings of the 21st Symposium on Mass Storage Systems*, Apr. 2004.

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

[37] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.

Capo: Recapitulating Storage for Virtual Desktops

Mohammad Shamma, Dutch T. Meyer, Jake Wires, Maria Ivanova,
Norman C. Hutchinson and Andrew Warfield
Department of Computer Science
University of British Columbia

Abstract

Shared storage underlies most enterprise VM deployments because it is an established technology that administrators are familiar with and because it good job of protecting data. However, shared storage is also very expensive to scale. This paper describes *Capo*[1], a transparent and persistent block request proxy for virtual machine disk images. Capo reduces the load on shared storage by using local disks as persistent caches, using multicast-based preloading to broadcast read results across a cluster, and by imposing *differential durability* – dividing a VM's file system into regions of varying writeback frequency. We motivate the system's design through the analysis of a week-long trace of 55 production virtual desktops and then describe and evaluate our implementation. Capo is particularly well suited for virtual desktop deployments, in which large numbers of VMs boot from a small number of "gold master" images and are refreshed on a periodic basis.

1 Introduction

The storage we trust is expensive. Fast and reliable data storage is something that organizations are prepared to pay a premium for, both in the capital costs of enterprise storage hardware and the operational costs of ensuring that important data is written to it.

Interestingly, the deployment of virtualization has inverted the historical imperative that systems be configured to "opt-in" to storing data on appropriate network shares instead of on less reliable locations such as local disks. While administrators used to have to work to configure applications to use enterprise storage, virtual environments simply store *everything* on it.

As such, these environments present the opposite problem: The requirement that virtual machine images be universally accessible, with high performance, to all physical hosts in a cluster has necessitated the deployment of SAN hardware in even modest virtualization deployments. The improved density and utilization afforded by virtualization allows systems to scale to large numbers of VMs; shared storage must scale proportionately to provide for them. This symptom is especially problematic for virtual desktops, where infrastructure is being deployed to host literally thousands of nearly identical VM images. A number of commercial virtual desktop systems now exist, and deployments suffer from a significant, if not dominant, cost for enterprise storage.

This paper argues that shared, central, storage *is* the correct approach for scalable virtual environments. It is trustworthy, relatively easy to manage, and simple to reason about. However, we believe that for applications such as virtual desktops, which involve large numbers of image clones, the majority of request load is redundant and can be effectively serviced by local, commodity disks within individual servers. Furthermore, we believe that the levels of durability provided by enterprise storage in these environments are in excess of what is necessary for large portions of desktop OS disk images.

The contributions of this paper are twofold: First, we validate our hypothesis through the analysis of a week-long trace of all storage traffic from a production deployment of 55 Windows Vista desktops in an executive and administrative office of a large public organization. Our results examine the opportunities that exist for caching data both within and across virtual desktop images. They also examine the breakdown of request workload within desktop filesystems.

Second, based on the analysis of this trace, we describe the design and implementation of *Capo*, a distributed persistent cache that aims to reduce aggregate load on shared storage in virtual desktop environments. Capo uses local server disks to provide persistent caching

[1] The name of our system is borrowed from the phrase *"Da Capo al coda"*, which is used in sheet music to indicate a brief return to the beginning of a piece, followed by a jump to the Coda, or conclusion. In sonatas, this "recapitulation" involves revisiting a similar, but sometimes different version of the main theme of the arrangement.

of VM images, and includes mechanisms to share and pre-load caches of gold master images across VMs and across hosts. Finally, Capo introduces a facility for *differential durability*, which allows administrators to selectively "opt out" of enterprise storage guarantees by relaxing the durability properties of subsets of a desktop's file system.

Virtual desktop systems have already taken advantage of several approaches to scale storage to large numbers of desktop machines. We begin in Section 2 by providing some brief initial background on these systems.

2 VDI Background

Virtual desktops represent the latest round in a decades-long oscillation between thin- and thick-client computing models. So-called Virtual Desktop Infrastructure (VDI) systems have emerged as a means of serving desktop computers from central, virtualized hardware. VDIs are being touted as a new compromise in a history of largely unsuccessful attempts to migrate desktop users onto thin clients, and the approach does provide a number of benefits. Giving users private virtual machines preserves their ability to customize their environment and interact with the system as they would a normal desktop computer. From the administration perspective, consolidating VMs onto central compute resources has the potential to reduce power consumption, allow location-transparent access, better protect private data, and ease software upgrades and maintenance.

Commercial VDI systems appear to be experiencing a degree of success: Gartner predicts that forty percent of all worldwide desktops—49 Million in total—will be virtualized by 2013 [17]. Today, the two major vendors of VDI systems, Citrix and VMWare, individually describe numerous case studies of active virtual desktop deployments of over 10,000 users. From a storage perspective, VDI systems have faced immediate challenges around space overheads and the ability to deploy and upgrade desktops over time. As background, this section describes how these problems are typically solved in existing architectures, as illustrated in Figure 1.

2.1 Copy-on-Write and Linked Clones

Operating system images are entire virtual disks, often tens of gigabytes each. A naive approach to supporting hundreds or thousands of virtual machines results in two immediate storage scalability problems. First, VMs must have isolated disk images, but maintaining individual copies of every single disk is impractical and consumes an enormous amount of space. Second, adding new users requires that images can be quickly duplicated without necessitating a complete copy.

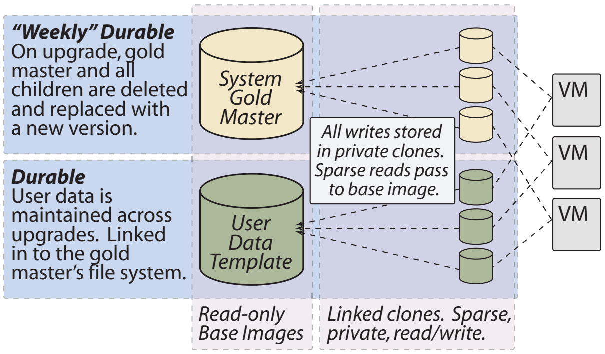

"Weekly" Durable
On upgrade, gold master and all children are deleted and replaced with a new version.
System Gold Master
Durable
User data is maintained across upgrades. Linked in to the gold master's file system.
User Data Template
All writes stored in private clones. Sparse reads pass to base image.
VM
VM
VM
Read-only Base Images
Linked clones. Sparse, private, read/write.

Figure 1: Typical image management in VDI systems.

This observation is not new; it has been a recurring challenge in virtualization. Existing VDI systems make use of VM-specific file formats such as Microsoft's VHD [14] and VMware's VMDK [22]. Both allow a sparse overlay image to be "chained" to a read-only base image (or gold master). As shown in Figure 1, modifications are written to private, per-VM overlays, and any data not in the overlay is read from the base image. In this manner, large numbers of virtual disks may share a single gold master. This approach consolidates common initial image data, and new images may be quickly cloned from a single gold master.

2.2 Image Updates and Periodic Rollback

Image chaining saves space and allows new images to be cloned from a gold master almost instantaneously. It is not a panacea though. Chained images immediately begin to diverge from the master version as VMs issue writes to them. One immediate problem with this divergence is the consumption of independent extra storage on a per-image basis. This divergence problem for storage consumption is typically addressed through the use of data deduplication [24, 6, 4].

For VDI, wasted storage is not the most pressing concern: block-level chaining means that patches and upgrades cannot be applied to the base image in a manner that merges and reconciles with the diverged clones. This means the ability to deploy new software or upgrades to a large number of VMs, which *was* initially provided from the single gold master is immediately lost.

The leading VDI offerings all solve this problem in a very similar way: They disallow users from persisting long term changes to the system image. When gold master images are first created and clones are deployed, the VDI system arranges images to isolate private user data (documents, settings, etc.) on separate storage from the system disk itself. As suggested initially in the Collective project [3], this approach allows a new gold master with updated software to be prepared and deployed to VMs simply by replacing the gold master, creating new (empty) clones, and throwing away the old version of the system disk along with all changes. This approach effectively "freshens" the underlying system image of all users periodically and ensures that all users are using a similar well-configured desktop. For the most part, it also means that users are unable to install additional, long-lived software within VDI images without support from administrators.

3 Virtual Desktop Trace

To better understand VDI workloads, we arranged to measure all block and file level activity from the a production VDI deployment for a one week period during the Summer of 2010. The deployment being studied is an office in a large public organization containing executive and administrative support staff. The deployment had been in production use for six months and includes 55 Windows Vista desktops, the organization is in the process of rolling out another 300 desktops this fall.

3.1 Methodology

We installed a Windows storage class driver into the base system image of the virtual desktop machines. The driver was written to record block read and write events to the virtual disks using the Microsoft Windows Software Trace Preprocessor (WPP). It recorded request size and virtual disk address. In 93% of cases we were also able to determine the file on which the access originated by following the OriginalFileObject pointer in the Windows I/O Request Packet (IRP) structure. To better contextualize this information, we also installed a driver at the filesystem level and recorded cache accesses, the application making each request, and the file flags for each file accessed. Our disk-level driver is written in 515 lines of C, while our file-level driver is 82 lines of C.

Logs from these drivers were written to a network share and collected on the Thursday following a full week of logging. In total we collected 75GB of logs in a compressed binary format. We then checked for corruption, missing logs, or missing events. Out of over 300 million entries we found a single anomalous write to a clearly invalid block address, which we removed. We could find no explanation for the event. In the rest of this section we present our analysis of this data. Unless otherwise specified, we will refer to block level accesses to a virtual disk and measure aggregate workloads in I/O operations per second (IOPS).

3.2 Our Virtual Desktop Environment

The environment we are studying is structured very much like the one described in Section 2. At the time our measurements were gathered it hosted 55 Microsoft Windows Vista virtual desktops with VMWare View, of which roughly 27 are in dedicated day-to-day use as the primary desktop. This small size is the primary limitation of our study, but we expect to measure considerably more as the installation grows. Furthermore, even at the current size it is possible to see considerable self-similarity among machines, as we will discuss.

End users work from Dell FX100 Zero thin clients, while VMs are served from HP BL490c G6 Blades running ESX Server. These servers connect to a Network Appliance 3170s over fiber channel, for booting from the SAN, and 10GigE, for VM disk images. System images are hosted via NFS on a 14 drive RAID group with 2 parity disks. The operating systems and applications are optimized for the virtual environment [20] and are pre-loaded with Firefox, Microsoft Office Enterprise, and Sophos Anti-Virus among other software. At the end of every Wednesday, a new system image is published to all users exactly as discussed in Section 2.2.

3.3 Analysis

We begin by asking, ***What are the day-to-day characteristics of VDI storage workloads?*** Figure 2 shows the entire study in I/O operations-per-second for the 24-hour period of each of the 7 full days recorded. There is a distinct peak load period between 8:30 and 9:30 every morning, as employees arrive at work. Three peaks in this period are highlighted in the figure and presented for expanded analysis in Table 1. The right-most column shows the applications responsible for the most disk I/O, excluding the system and services. Most days, Firefox and the virus scanner are very active in this period, we also see Thunderbird, Pidgin, and Microsoft Outlook frequently. We were surprised to see the Search Indexer active as well, because we were told its background scanning task had been disabled to reduce I/O consumption. Our best guess is that it was invoked manually.

We measured the write to read percentages for both IOps and throughput, which is useful in characterizing the workload. Our workload is write-heavy in IOps, and read-heavy in throughput, both by approximately two-to-one. We then measured the percentage of VMs which contributed at least 5% of the peak workload, to determine if peaks were caused by multiple VMs or by a few outliers. In most cases, it is the former; however, the peak in slice 4 was caused primarily by 4 VMs. The column titled "Dup. reads" illustrates the potential for caching. We present two numbers. The left-most is the percent-

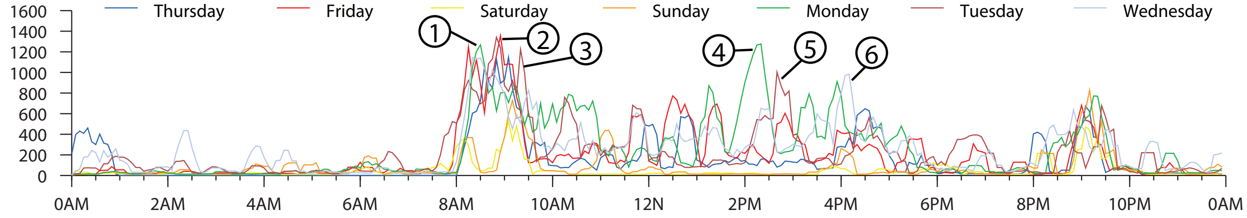


Figure 2: Activity measured in I/O operations over each of the 7 days.

	Time Period	Write % (IOps / Thpt)	% of VMs ($\geq$ 5%)	Dup. reads (VM / Clust.)	Top Applications by IOps (excludes system and scvhost)
1	Mon. 8:30am-8:45am	50% / 22%	26%	81% / 91%	Search Indexer, Firefox, Sophos
2	Fri. 8:30am-9:00am	52% / 22%	29%	88% / 97%	Firefox, Search Indexer, Sophos
3	Tue. 9:15am-9:30am	64% / 43%	29%	78% / 99%	Defrag, Firefox, Search Indexer
4	Mon. 2:00pm-2:30pm	62% / 41%	7%	59% / 99%	Firefox, Pidgin, Sophos
5	Tue. 2:40pm-3:00pm	69% / 52%	26%	77% / 97%	Firefox, Defrag, Pidgin
6	Wed. 4:00pm-4:15pm	60% / 37%	26%	99% / >99%	Firefox, Pidgin, Sophos

Table 1: Points of interest in Figure 2.

age of reads that have been previously seen by that VM over the course of the trace. With a large enough cache, we could potentially absorb *all* these reads. The right-hand column presents the same measure, but imagines that caching could be shared across all VMs in the cluster. Slice 4 stands out for having an unusually low duplicate read rate for VMs, but a very high rate across the cluster as a whole. We investigated and found that two very active VMs had duplicate read rates of 26% and 30%. By including the least beneficial 38%, 15% and 4% of VMs, we could reach duplicate read rates of 40%, 60% and 90% respectively. From this we conclude that you can achieve significant improvements with caching, possibly even by sharing caches, but that some benefits may require careful selection of the VMs in question.

Lunch, dinner, and late nights are periods of relative inactivity, as are the weekends. Late afternoon peaks are sporadic, but reach loads nearly as high as the morning. One such peak, marked 4 in Figure 2 and Table 1, was caused by a relatively small number of machines engaged in heavy browsing activity. This is not the norm, as all other peaks occur when more than a quarter of the VMs are significantly active. This is clear in Figure 3, which shows a CDF of VMs by their contribution to the total workload for each peak. These peak load periods are particularly important, because they define the hardware necessary to service the workload without disruption. We conclude that, ***VDI workloads are defined by their peaks, and those peaks usually occur at times of common activity among many VMs.*** In Sections 4.1 and 4.2 we take advantage of this fact to improve performance in VDI environments.

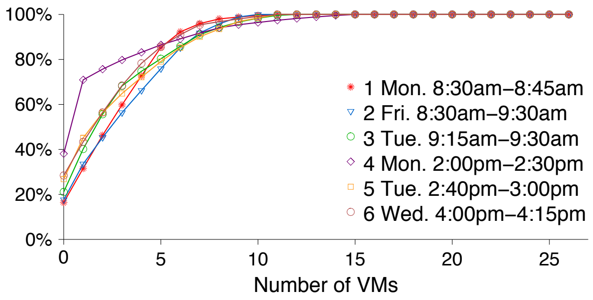


Figure 3: Contributors to each of the major workload peaks.

Next, we ask, ***How can we characterize the I/O requests?*** In Figure 4 we show the entire workload by both request count and size. We differentiate reads from writes, and also isolate each request by its target in the file system namespace. The workload is 65% writes, which account for 35% of the throughput, versus 35% of reads accounting for 65% of the throughput. Metadata operations account for large portion of the requests; unfortunately we cannot determine how these modifications relate to the namespace. Directories typically managed by the operating system, such as `\Windows` and `\Program files` are also frequently accessed. There are fewer accesses to user directories and temporary files; most of the latter are to `\Temporary Internet Files`, as opposed to `\Windows\Temp`. These findings contrast those of Vogels who's study showed that 93% of *file-level* modification occurred in `\User` directories [23]. We conclude that, ***while a wide range of***

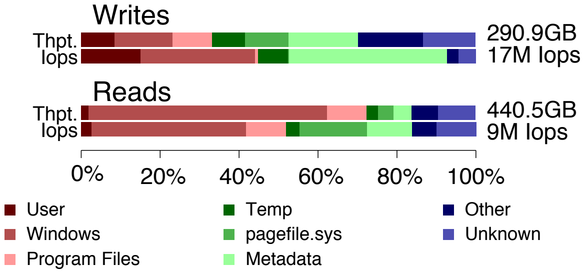


Figure 4: Size and amount of block level writes by file system path.

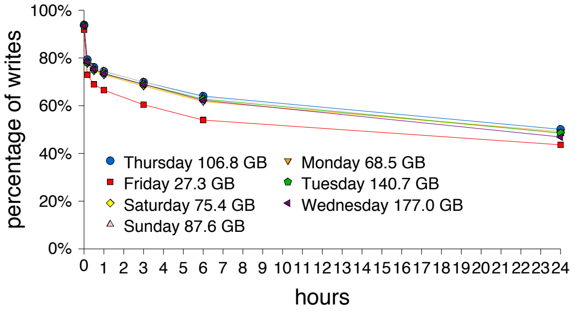


Figure 5: Percentage of bytes that need to be written to the server if writes are held back for different time periods. This is lower than the original volume of writes due to the elimination of rewrites.

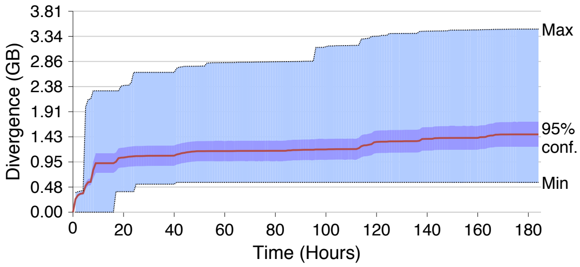


Figure 6: Bytes of disk diverging from the gold master.

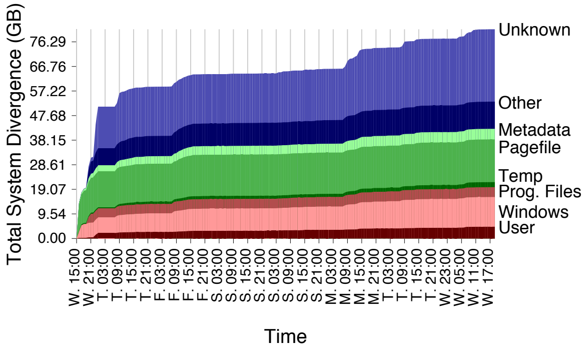


Figure 7: Total divergence versus time for each namespace category.

the namespace is accessed, it is not accessed uniformly, and access to data directly managed by users is rare. We will revisit this observation in Section 4.3.4.

Since our workload is write-heavy, we next ask, ***how are these writes organized in time?*** Figure 5 shows the percentage of disk writes that overwrite recently written data, for time intervals ranging from 10 seconds to a whole day. We have included results from each of the seven days to underscore how consistent the results are. In a short time span, just 10 seconds, 8% of bytes that are written are written again. This rate increases to 20%-30% in 10 minute periods and ranges between 50%-55% for twenty-four hour periods. From this we conclude that, ***Considerable system-wide effort is spent on data with a high modification rate.*** We show how this can used to our advantage in Section 4.3.

Since VMs typically use disk images chained from a gold master, we are interested in the rate at which the overlay image diverges from the original image. We therefore ask, ***At what frequency do we observe the first write to a sector?*** Figure 6 plots this data for the average VM, as well as the most and least divergent VM, over the entire study. Within 24 hours, most VMs hit a near plateau in their divergence, around 1GB. Over time this does increase, but slowly. A smaller set of VMs do diverge more quickly and significantly, but they are far from the 95% confidence interval. We conclude that, ***there is significant shared data between VMs, even after several days of divergence.***

Naturally, we do not expect divergent writes to occur uniformly, so we pose a question: ***Where in the namespace do divergent writes occur, and does this change over time?*** Figure 7 plots the cumulative divergence for each VM in the cluster, and divides that total among various components of the namespace. One can observe that the pagefile diverges immediately, then remains a constant size over time, as does the system metadata. Both these files are bounded in size. Meanwhile writes to `\Windows` and areas of the disk we cannot associate with any file continue to grow over the full week of the study. We conclude that, ***While writes occur everywhere in the namespace, they exhibit significant trends when categorized according to the destination.***

3.4 Summary

While there is more to say about this workload and those of VDI environments in general, the observations in this section are valuable. Summarizing our observations from the trace data:

- VDI workloads are defined by their peaks, and those peaks usually occur at times of common activity among many VMs
- While a wide range of the namespace is accessed, it is not accessed uniformly, and access to data directly managed by users is rare
- Considerable system-wide effort is spent on data with a high modification rate
- There is significant shared data between VMs, even after several days of divergence
- While writes occur everywhere in the namespace, they exhibit significant trends when categorized according to the destination

These observations taken together suggest that additional caching, combined with an awareness of namespace organization might resolve the performance challenges that we have observed. The following section builds on the observations and analysis presented here, and describes the architecture of Capo.

4 Architecture

The trace analysis in Section 3 suggests that caching below the individual VMs may be effective in resolving the demand peaks that we observed. In this section we describe Capo, including its three major components:

1. A single-host cache which eliminates redundant reads and writes from virtual desktops hosted on the same server.
2. A multi-host cache preloader which eliminates redundant reads from virtual desktops hosted on different servers.
3. A component that supports *differential durability*, which modifies cache coherency based on the location in the namespace of the affected file.

Figure 8 shows the overall architecture of Capo. Capo exists as a layer within the virtual machine monitor (VMM) which supports the individual desktop VMs. The figure depicts each host including a Local Persistent Cache which is stored on the local disk of the host machine and is described in Section 4.1. Spanning all of the hosts is the Transparent Multi-host Prefetch component which optimistically preloads data accessed by one host into the local caches of the other hosts. It is described in Section 4.2. The Durability Map component supports the wide variation in the durability requirements of the various components of the file system. It is described in Section 4.3.

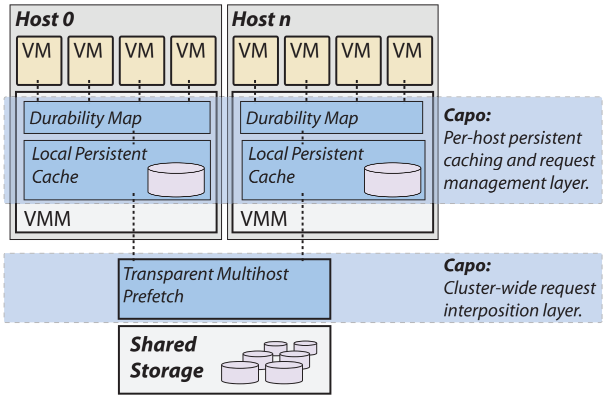


Figure 8: The Major Architectural Components of Capo.

4.1 Local Persistent Cache

All VDI deployments rely on central enterprise storage that provides high availability, durability, and reliability. The servers that host virtual desktops are also configured with local disk storage which consists of cheap COTS disk drives with comparably lower reliability but higher aggregate I/O bandwidth.

The trace-based analysis of our local VDI deployment suggests that a cache shared between multiple virtual desktops might be very effective. As shown in Figure 3, there is significant overlap between the top applications executed on different virtual machines. Table 1 refines this and indicates that aggressive caching can yield very high read hit rates. Also, as shown in Figure 5, a significant fraction of data is overwritten very quickly. Therefore, as depicted in Figure 8, each server machine that hosts virtual desktops uses its local disk as a persistent cache. A key goal for Capo is to provide an appropriate level of durability for all data while taking advantage of the higher aggregate bandwidth available to local disk. The level of durability achieved depends on the caching policy in place.

4.1.1 Caching Policies

The cache supports two consistency policies: write-through and write-back. These policies are enforced at disk image granularity. Each of these policies represents a different tradeoff between virtual disk consistency and overall system performance.

The *write-through* policy provides the highest level of consistency guarantees a machine would expect from a block device. In this policy the cache replicates writes to both the centralized storage and the local cache. Write requests are not acknowledged until they hit both disks. This policy relieves the centralized storage from serving reads to blocks that have been previously read from or written to. The drawback of this policy is that write requests must be sent across the network, consuming network bandwidth and increasing both the load on the centralized storage and the client's perceived write request latency.

The *write-back* policy delays pushing updates to disk blocks by caching writes locally in the write cache. Updates are pushed to the central storage in a crash consistent manner at a per-virtual-disk configurable frequency. The choice of write-back frequency trades off system performance and durability of disk contents in case of a failure. A high update frequency minimizes the amount of data loss in case of the failure of the local disk, while a lower frequency enhances overall system performance by coalescing writes in the local cache.

4.1.2 Design and Implementation

The local persistent cache is implemented as an extension to the publicly available XenServer 5.6 release. It runs in Xen's "domain 0" VM and interposes on the block request path below virtual machines. Cached data is stored as sparse image files in a Linux file system. Each virtual disk's cache consists of either two or three components, shown in Figure 9: a read store, write store, and possibly a snapshot store. Each of these components is represented using a data file and a bitmap in the persistent cache. The bitmap's purpose is to identify which sectors of the corresponding data file are valid. Writing a sector to a cache component involves writing the sector's data to the data file and setting the sector's corresponding bits in the on-disk bitmap.

Write requests are satisfied by writing their data sectors to the cache's write store. When the cache is set to write-through, the sectors are also written concurrently to the centralized storage. Read requests are satisfied by first checking the write store, then the snapshot if it exists, and finally the read store. At each layer, if a sector is valid as specified in the bitmap, the data can be returned immediately from that layer. If none of the stores contain valid data, the sectors are read from the centralized storage, written to the read store, and returned to the client VM.

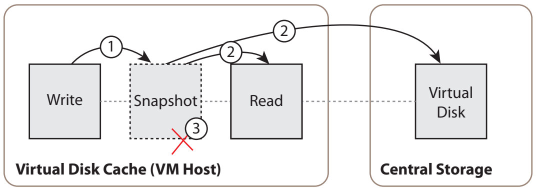


Figure 9: Capo's virtual disk cache components and snapshot procedure.

The snapshot mechanism in the cache works in tandem with a transactional update mechanism in the backend storage to ensure crash consistent updates to remote disk images when operating in write-back mode. Pushing updates to the backend storage involves three steps, as shown in Figure 9. First, a write cache snapshot is created by pausing the request stream momentarily and moving the contents of the cache's write store to the snapshot store. Secondly, the snapshot contents are applied transactionally to the centralized storage and to the read cache concurrently. Finally, after the snapshot updates have been applied, the snapshot store is cleared.

The cache is implemented as a user-level shared library that interposes on I/O calls, specifically the glibc and libaio I/O and file management operations. Due to the relative sizes of the disks in virtual desktops (around 10GB) and the disks in the physical machine which hosts them (greater than 1TB) and the amount of sharing between virtual desktops (see Figure 6), we can easily support hundreds of virtual desktops on a server without worrying about overfilling the cache. In our implementation, when the cache does fill, we simply throw it away and start again.

4.2 Multi-host Cache Preload

Capo's local persistent cache goes a long way towards eliminating redundant read requests on individual machines. But as growing VM deployments lead to larger numbers of physical hosts, redundant reads across these hosts place additional burden on central servers. Further scalability improvements can be attained in this case by multicasting common data to all hosts simultaneously rather than to each host individually.

To this end, we have developed a multicast cache preloader for local caches. The preloader is completely lock-free and requires no modifications to existing centralized servers. It consists of a service which observes network traffic to and from the central storage server. Clients on each host contact the service and register watches for files which are determined to be good candidates for preloading. The service captures any reads made to these files and distributes the results to all subscribed clients via multicast. In this way, the first host to read common data essentially prefetches it for all other hosts.

4.2.1 Design and Implementation

Our initial design for the preload server was to use a mirror port on the central storage server to monitor network traffic. As in *Ditto* [5], our server captured raw network packets and reconstructed TCP flows to extract relevant data (in our case, NFS requests and responses). When deploying this solution, however, we observed significant packet loss between the mirror port on the filer and our server, and since a single packet loss is enough to corrupt an entire NFS request or response, we were missing many opportunities to preload data.

Our second, and current, design employs a user-level NFS proxy that sits between the clients and the filer. NFS requests and responses are routed through the proxy, and the proxy identifies data that should be preloaded into other local persistent caches. This increases the latency of filer requests somewhat, but avoids all of the issues with packet capture.

In the current implementation, NFS clients are left unmodified. Instead, a single preload client runs on each physical host. On startup, these processes register watches with the server for files known to be shared across hosts. Because this data is predominately read-only, no synchronization is required when multicast clients update the local caches. When the preload server observes reads to files being watched by clients, it multicasts the responses to all clients.

Because NFS clients are unmodified, reads of shared files result in two responses: the unicast response to the original requester, and a second multicast response to all subscribed clients. This leads to an increase in overall read bandwidth consumption from the proxy to the clients, but reduces the load on the storage server. The redundant unicast response could easily be avoided by making NFS clients aware of the multicast service.

We also currently prioritize unicast responses over multicast responses. This limits the latency overhead seen by NFS clients while delaying preloading on other clients, making it slightly more likely that they will submit unicast requests for the same data. With modified NFS clients, we could more viably prioritize multicast responses, improving the efficacy of preloading.

The preload server sends a significant amount of traffic over a number of multicast sessions, and has exposed problems with the support for multicast in some modern switches. On some of the switches that we have experimented against, multicast packets appear to consume a disproportionately large amount of resources. As a result, even relatively low-throughput multicast traffic has resulted in packet drops with detrimental consequences for concurrent TCP connections. The results can be dramatic: early experiments with completely unthrottled multicast traffic resulted in NFS throughput drops from 100MB/sec to 3MB/sec.

To address this, we have implemented a rudimentary, adaptive flow-control protocol, similar to one described in *SnowFlock* [13]. Each packet sent by the server is associated with an epoch. The server periodically updates the epoch number, and when clients notice a new epoch, they send a message indicating the number of packets successfully received during the previous epoch. In this way the server gets feedback about packet drop rates and is able to vary transmission rates accordingly. An additive increase/multiplicative decrease scheme with aggressive back-off has produced reasonable results in our benchmarks.

This flow-control protocol – and preloading in general – is strictly best-effort: no work is wasted trying to retransmit dropped multicast packets. If the preload clients fail to receive multicast updates for required data, it will eventually be fetched via the conventional unicast path.

The client logic for deciding which files to preload is simplified by a few basic design principles. We assume that, given a number of VMs derived from a common master image, reads of the base image made by any individual VM will likely be duplicated by all VMs. That is, while the disks belonging to derived VMs will tend to diverge as the VMs age, the common portion of these disks will likely be read by all or none of the VMs. Thus if the multicast server observes any read of a common file, it is worth sending this data to all hosts on which Capo is caching this file. By the same assumption, multicast clients do not pro-actively request data from the server, as they are not in a position to know which portions of files will be read by VMs.

4.3 Differential Durability

Major VDI providers have all adopted the software update strategy proposed in The Collective [3], where user directories are isolated from the rest of the file system. Modifications made to files in the user directories must be durable; users depend on these changes. Capo therefore uses write-through caching on these directories, propagating all changed blocks immediately to the centralized storage servers. Any modifications to the system image can then be performed on all VMs in one step by completely replacing the system images in the entire pool, leaving the user's data unmodified. This impacts durability—any writes to the system portion (e.g., by updating the registry or installing software) will be lost. In this section we use and extend this notion to optimize for our write-heavy workload.

4.3.1 Write-Back Period

As mentioned in Section 2.2, VDI deployments manage system data centrally, regularly replacing the system data seen by each virtual desktop with a clean updated version. While users are allowed to make changes to their system data, these changes are not guaranteed to be durable. Writes to the `\Program Files` directory as part of an application install process, for example, represent work done by a user, but a software installation could easily be repeated if a failure caused this to be nec-

Path	Policy
\Program Files\	write-back
\WINDOWS\	write-back
\Users\ProgramData\VMware\VDM\logs	write-back
\Users\$USER$\ntuser.dat	write-back
\Users\$USER$\AppData\local	write-back
\Users\$USER$\AppData\roaming	write-back
\pagefile.sys	no-write-back
\ProgramData\Sophos	no-write-back
\Temp\	no-write-back
\Users\$USER$\AppData\Local\Microsoft\Windows\Temporary Internet Files	no-write-back
Everything else, including user data and FileSystem metadata	write-through

Table 2: Sample cache-coherency policies applied as part of durability optimization.

essary. It might be acceptable if the loss of such effort was limited to, for example, an hour or even a day. We can set our write-back period for such partially durable files to a corresponding length of time.

4.3.2 Extending Partial Durability to User files

While much of the data on the User volume is important to the user and must have maximum durability, Windows, in particular, places some files containing system data in the User volume. Examples include log files, the user portion of the Windows registry, and the local and roaming profiles containing per-application configuration settings. Table 2 shows some paths on User volumes in Windows that can reasonably be cached with a write-back policy and a relatively long write-back period.

4.3.3 Eliminating Write-Back

There are some system files that need not be durably stored at all. These include files that are discarded on system restarts or can easily be reconstructed if lost. Writes to the pagefile, for example, represent nearly a tenth of the total throughput to centralized storage in our workload. These writes consume valuable storage and network bandwidth, but since the pagefile is discarded on system restart, durably storing this data provides no benefit. The additional durability obtained by transmitting these writes over a congested network to store them on highly redundant centralized storage provides no value because this data fate-shares with the local host machine and its disk. Many temporary files are used in the same way, requiring persistent storage only as long as the VM is running.

We store this data to local disk only, assigning it a write-back cache policy with an infinitely long write-back period. In the event of a hardware crash on a physical host, the VM will be forced to reboot, and the data can be discarded.

4.3.4 Design

Initially, we approached the problem of mapping these policies to write requests as one of request *tagging*, in which a driver installed on each virtual desktop would provide hints to the local cache about each write. While this approach is flexible and powerful, maintaining the correct consistency between file and filesystem metadata (much of which appears as opaque writes to the Master File Table in NTFS) under different policies is challenging. Instead, we have developed a simpler and better performing approach using existing filesystem features.

The path-based policies we use in our experiments can be seen in Table 2; naturally, these may be customized by an administrator. We provide these policies to a disk optimization tool that we run when creating a virtual machine image. The optimization tool also takes a populated and configured base disk image. For each of the two less-durable policies, it takes the given path and moves the existing data to one of two newly-created NTFS file systems dedicated to that policy. It then replaces the path in the original file system with a reparse point (Window's analogue of a symbolic link) to the migrated data. This transforms the single file system into three file systems with the same original logical view. Each of the three file systems are placed on a volume with the appropriate policy provided by the local disk cache. This technique is similar to the view synthesis in Ventana [18], though we are the first to apply the technique with a local cache to optimize performance.

We appreciate that applying different consistency policies to files in a single logical file system may be controversial. The risk in doing so is that a crash or hardware failure results in a dependency between a file that is preserved and a file that is lost. Such a state could lead to instability; however, we are aware of no dependencies crossing from files with high durability requirements to those with lower durability requirements in practice. Further, we observe that this threat already exists in the production environment we studied, which overwrites system images with a common shared image on a weekly basis.

5 Evaluation

To evaluate the effectiveness of Capo, we first consider how effective differential durability is at reducing write load from unimportant regions of disk. Next, we show the storage reduction achieved by Capo with eleven concurrent users synthesizing active desktop workloads. Finally we show the storage reduction achieved by Capo by replaying I/O logs gathered from a production system (see Section 3) under different caching policies.

5.1 Differential Durability

This section describes several microbenchmarks that evaluate the effectiveness of differential durability in isolation of other features and provide a clearer mapping of end-user activity to observed writes. We applied the policies in Table 2 to several realistic desktop workloads. For each, we measured the portion of write requests that would fall under each policy category.

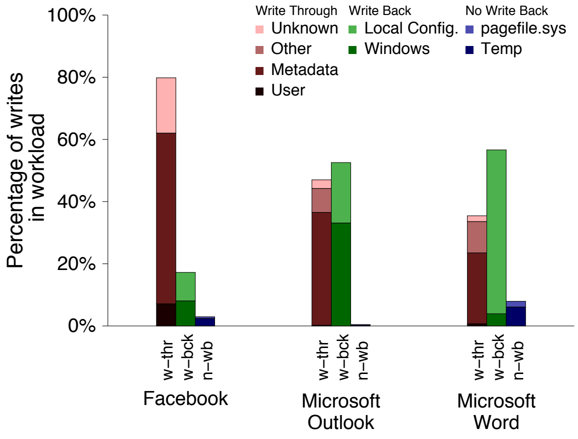


Figure 10: Percentage of writes in three microbenchmarks organized by governing cache-coherence policy.

5.1.1 Web Workload

Our web workload is intended to capture a short burst of web activity. The user opens www.facebook.com with Microsoft Internet Explorer, logs in, and posts a brief message to their account. They then log off and close the browser. The entire task lasts less than a minute. The workload consists of 8MB (43.6% by count) of writes and 25.3MB (56.4% by count) of reads.

A breakdown of writes by their associated policy for each workload is shown in Figure 10. In this short workload only a small, but non-trivial improvement can be made. Local configuration changes such as registry, temp file, and cache updates need not be written immediately, removing or delaying just over 20% of the operations.

5.1.2 Email Workload

Our email workload is based on Microsoft Outlook. The user sends emails to a server we have configured to automatically reply to every message by sending back an identical message. Ten emails are sent and received in succession before the test ends. The workload consists of 63MB (39% by count) of writes and 148MB (61% by count) of reads.

Here the improvement is much more substantial. Although very few writes can be stored to local disk indefinitely, over half can be delayed in writing to centralized storage. This is due to Outlook's caching behavior, which makes heavy use of the system and application data folders. Emails in the .pst file are included in the user category. It is worth noting that many files in the windows and application data are obvious temp files, but did not match our current policies. With more careful tuning, the policies could be further optimized for this workload.

5.1.3 Application Workload

Our application workload is intended to simulate a simple editing task. We open Microsoft Word and create a new document. We also open www.wikipedia.org in Microsoft Internet Explorer. We then proceed to navigate to 10 random Wikipedia pages in turn, and copy the first paragraph of each into our word document, saving the document each time. Finally, we close both programs. The workload consists of 120MB (20.0% by count) of writes and 406MB (80.0% by count) of reads.

Viewing many small pages creates a large number of small writes to temporary files and memory pressure[2] increases the pagefile usage. Both programs write significantly to system folders, leaving less than 36% of the workload to be issued as write-through.

5.2 Multi-host Cache Preload

To evaluate the effectiveness of Capo's multi-host prefetching we boot three Windows XP VMs on three different hosts. The experiment first fully boots one VM

[2] The guest was running Windows Vista with 1GB of RAM, 25% higher than the XenDesktop recommended minimum.

before booting the other two VMs concurrently, with the intention of demonstrating that the reads triggered by the boot on the first host are sufficient to achieve a savings for the later boots.

Figure 11 shows the read workload observed at the server in three different cache configurations: *No cache*, *Write-through* and *Write-through with multi-host preload*. Notice that the read workload for booting the two VMs is roughly double that of booting a single VM for both the *no cache* and *write-through* configurations. On the other hand *write-through with multi-host prefetching* almost eliminates the workload due to booting the two later machines.

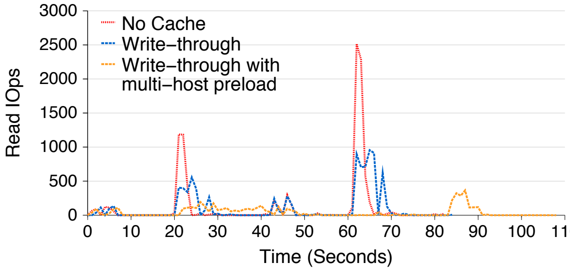


Figure 11: Read IOps per second for booting three VMs on three different hosts.

5.3 Synthetic Workload

To evaluate Capo as a whole, we arranged to simulate a set of (very) active desktop users, performing similar workloads to those seen in the trace. Figure 13 shows the results of request traffic hitting both the local caches (in aggregate across all images) and the filer, while 11 users actively use a variety of office and web-based applications.

First, note that the load in this case is higher than any of the peaks seen in the trace data. This workload represents a higher level of aggregate storage activity than was ever seen in the production environment. Second, observe that despite being configured conservatively for complete write back, Capo reduces all peaks in the storage request load.

5.4 Trace Replay

To evaluate the benefits of deploying Capo in a real world setting, we replay the collected I/O traces (see Section 3) using different disk caching policies. The next sections describe our experimental setting, analyze our replayer fidelity, and present the replay results.

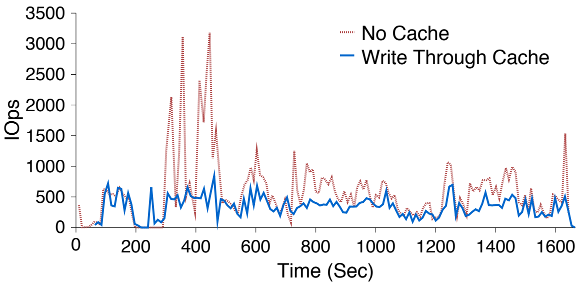


Figure 13: IOps per second for a workload of 11 Windows users on a XenCenter Cluster.

5.4.1 Experimental Setting

The test environment consists of four physical machines which serve as hosts for the virtual machines that replay requests from the recorded trace, and a filer to serve as a backend storage for these virtual machines' disks. The filer runs Linux's default kernel NFS server to host an XFS volume built on top of a RAID 0 consisting of six disks. The host machines run XenServer 5.6 and store their local caches in an ext3 volume on top of a RAID configuration similar to that of the filer. The machines are connected using a 1Gb Ethernet switch.

We replayed the workload of each desktop for which we had collected traces in a distinct virtual machine on one of the XenServer hosts. As it is impractical to replay the entire week's trace for each configuration, we choose to focus on the six peak regions identified in Section 3.

Entirely isolating our analysis to the peak regions would start each replay with an empty cache. Instead, we accurately recreated the state the cache would be in at the start of each region by priming it with the data from whole trace up to that point. This includes any write-back blocks that would have been pending. The write-back interval was set to ten minutes for the write-back and differential durability policies.

5.4.2 Replay Fidelity

Both the hosts and the storage server in our replay experiment are different from those in the original system from which we collected the traces. We satisfied ourselves that the replay is representative by measuring the observed load during a simple replay without any caching. Figure 12 plots the fourth selected time period's I/O operations per second as observed at a number of different points in the I/O stack of the experimental environment.

The *Trace* line represents the aggregate workload as observed in the original trace. The *Replay* line represents the rate at which the replayer issues I/O requests to the system as observed at the replay clients. These two lines

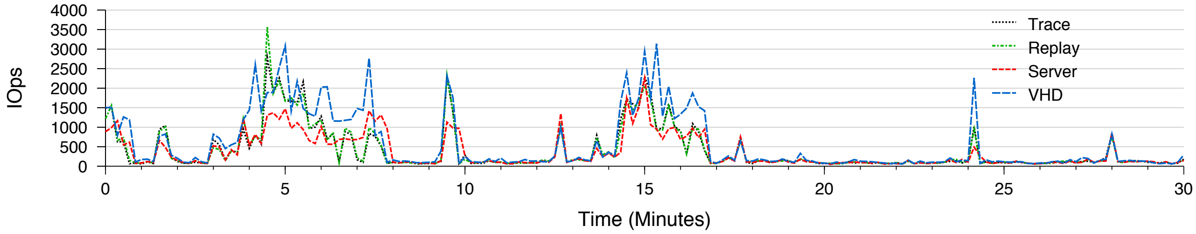


Figure 12: Replay fidelity and resulting load on the server.

Time Period	Peak IOps / Reduction in peak IOps compared to No Cache configuration							
	No Cache		Write Through		Write Back		Differential	
	Peak	Total	Peak	Total	Peak	Total	Peak	Total
1	2307 / 100%	262590 / 100%	893 / 38%	155757 / 59%	670 / 29%	67798 / 25%	712 / 30%	91877 / 34%
2	2516 / 100%	561894 / 100%	937 / 37%	319936 / 56%	671 / 26%	113184 / 20%	903 / 35%	161737 / 28%
3	1302 / 100%	143468 / 100%	876 / 67%	126049 / 87%	595 / 45%	43455 / 30%	802 / 61%	84044 / 58%
4	1887 / 100%	450914 / 100%	910 / 48%	334089 / 74%	595 / 31%	131064 / 29%	849 / 44%	271529 / 60%
5	1214 / 100%	159736 / 100%	890 / 73%	141656 / 88%	704 / 57%	45868 / 28%	841 / 69%	75500 / 47%
6	2185 / 100%	72082 / 100%	1155 / 52%	66668 / 92%	910 / 41%	29086 / 40%	1368 / 62%	42895 / 59%
avg	100%	100%	52.5%	76%	38.1%	28.6%	50.1%	47.6%

Table 3: Peak and Total IOps workload observed at the file server during the replay of time periods of interest under different caching policies. Each peak or total IOps value is followed by its ratio relative to its corresponding value observed with no cache deployed. The last row represents the average reduction of the metric across the six time periods of interest.

are almost indistinguishable in the figure which indicates that the timing of our replayer is accurate.

The *Server* line represents the load observed at the server. Notice that this load is lighter than the aggregate trace load, largely due to coalescing requests in the storage stack of the XenServer hosts. The *VHD* line represents the load observed at the server when image files are stored in the Microsoft VHD format. Notice that VHD adds significant overhead to the workload; most of this overhead is due to meta data management.

We draw two observations from this evaluation. First, our replay client is able to match the request issue rate of the original trace with high fidelity. Second, because of transformations that result from both the XenServer storage stack and the underlying VM image format, the load experienced at the storage target may be dramatically different from that measured at the client. In evaluating our cache under replay in the next subsection, we first replay with no caching involved to establish a baseline load at the filer, and then compare caching configurations to this baseline.

5.4.3 Replay Results

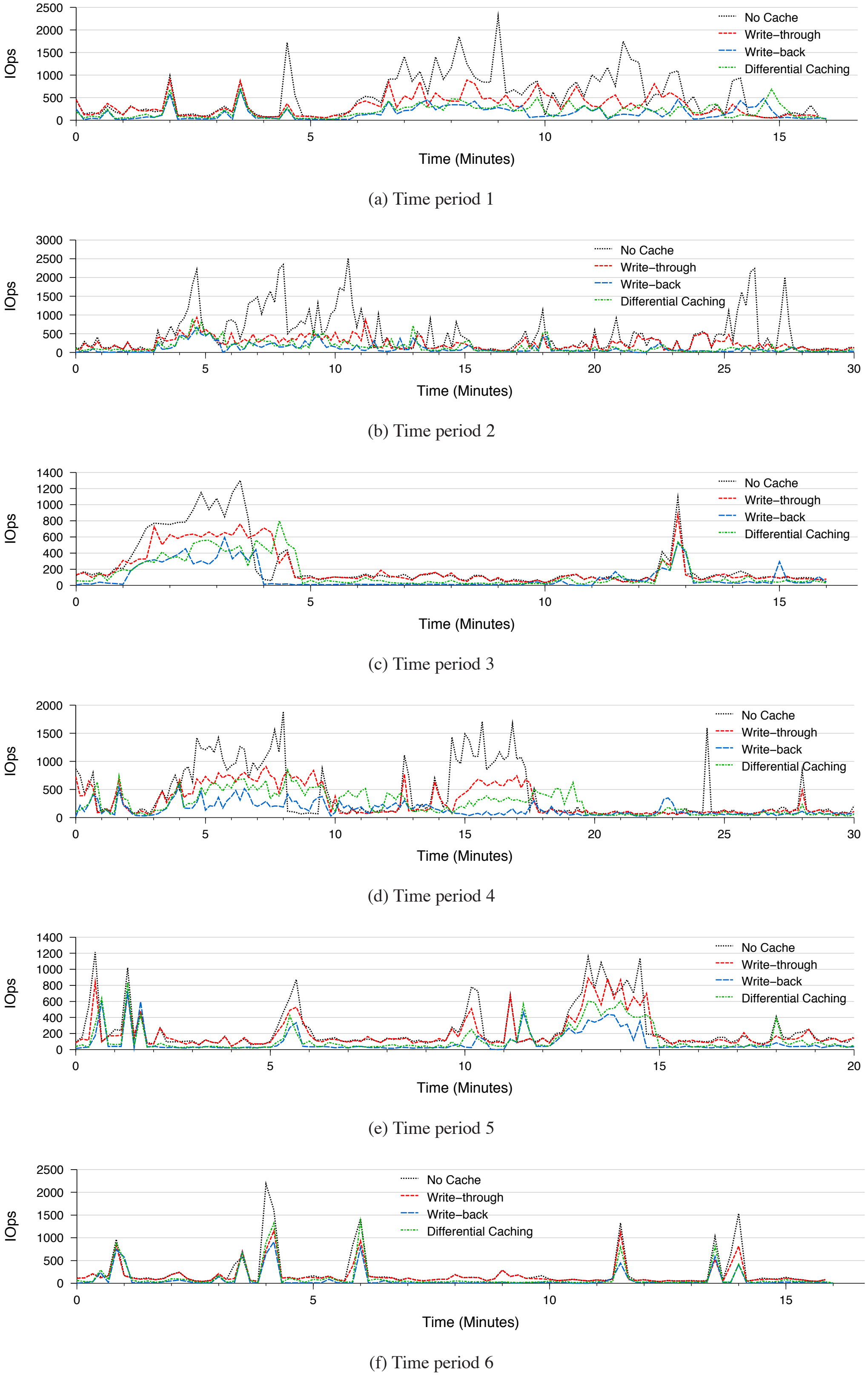
We replayed the 6 periods of intense workload identified in Figure 2 using four different cache configurations. These cache configurations are no cache, write-through, write-back and differential. Figure 14 plots the IOps observed at the server using each configuration. As expected, differential durability represents a compromise between the load reduction realized by write-back and completely protecting important user data.

Table 3 summarizes the peak and total I/O workload reductions for the time periods of interest. The write-back policy applied to the entire disk was the best in reducing I/O workload. On average it reduced the peak and total I/O workload to 38.1% and 28.6% of that without any caching in place. The differential durability policy goes further to protect user files, and still reduced the peak and total I/O workload down to 50.1% and 47.6% on average. Finally, as expected the write-through policy had the worst average peak and total workload reductions of 52.5% and 76%.

6 Related Work

The caching component of Capo is most closely related to the ITC [19], Andrew [10], and Coda [12] file systems which utilize the local disk as a cache for whole files retrieved from servers. The Cedar file system [21] allows users to share immutable files over the network; by only supporting immutable files Cedar eliminates the need for cache consistency management.

Unlike these distributed file system caches, Capo operates at the block level. Cache consistency management



(a) Time period 1

(b) Time period 2

(c) Time period 3

(d) Time period 4

(e) Time period 5

(f) Time period 6

Figure 14: IOps per second observed at the filer for replays of selected periods of interest under different cache configurations.

is simplified by the fact that each virtual disk has a single writer and copy-on-write is used to prevent updating shared data. As in Cedar, shared data is always immutable.

Fs-cache [11] and iCache [9] are systems that, like Capo, implement block-level caching for remote storage systems: a file system in the case of Fs-cache and an iSCSI target in the case of iCache. Capo extends the basic block caches of these systems using a host cache shared by all the VMs on a host, the multi-host prefetcher, and differential durability for files. All of these features are inspired by our target environment of supporting virtual desktops.

Capo's use of write-back caching reduces the demand placed on the central storage facility in a manner similar to that of Everest [15]. Where Everest replicates offloaded write requests to tolerate disk failures, Capo uses a technique similar to Snapmirror [16] to periodically push self-consistent updates across the network for data that is cached in write-back mode.

Other researchers have studied the performance of storage in virtualized environments. In particular, Gulati et al. [7] study the storage demands of enterprise applications in virtualized environments. In contrast, our study of virtual desktops provides insight into the unique characteristics of this emerging use of virtualization.

SnowFlock [13] provides a fork abstraction to instantaneously replicate stateful virtual machines to scale up computations in the cloud easily. Similar to our multi-host cache preloader, SnowFlock uses multicasting to replicate the persistent (disk) and non-persistent (memory) state of the cloned virtual machines.

Agrawal et al. [1] and Bolosky et al. [2] collect and analyze snapshots of Desktop machine's file system metadata over long periods of time. This kind of analysis restricts I/O workload analysis to mean estimates and doesn't capture the dynamic characteristics of Desktop I/O such as burstiness. In this work we focus on capturing detailed block level I/O operations to better understand the variation of Desktop I/O workloads in time.

Lithium [8] gives up centralization in favor of distribution to provide scalable storage for virtual machines. To improve availability of data, Lithium replicates disk updates to remote hosts either synchronously or lazily (eventual consistency). These two replication policies are synonymous to Capo's write-through and write-back caching policies. However, Lithium's treatment of replication consistency is more complicated due to its distributed nature.

7 Conclusion

Enterprise storage provides considerable benefit to virtual environments. However, for applications such as virtual desktops, which involve large numbers of nearly identical images running concurrently, a large portion of the request load placed on shared storage is unnecessary. After analyzing a one-week trace of a production VDI deployment, we presented Capo, a distributed and persistent cache which reduces the aggregate load placed on shared storage. Capo uses local disks on individual physical servers to cache image contents for the VMs being hosted. It includes mechanisms to share common cached base images across VMs, and to prefetch caches across physical hosts. In addition, Capo supports a configurable degree of differential durability, allowing administrators to relax the durability properties and the associated write load of less-important subsets of a VM's file system.

Acknowledgements

We thank our shepherd Greg Ganger, the anonymous reviewers, and Brendan Cully for their valuable feedback that helped improve this paper. We also thank the University of British Columbia's IT department, particularly Brent Dunington and Lois Cumming, for allowing us to collect I/O traces from one of their VDI deployments. We would also like to thank MITACS, NSERC, Citrix and IBM CAS who have provided support for the students involved in this work.

References

[1] N. Agrawal, W. Bolosky, J. Douceur, and J. Lorch. A five-year study of file-system metadata. *ACM Transactions on Storage (TOS)*, 3(3):9, 2007.

[2] W. Bolosky, J. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 34–43. ACM, 2000.

[3] R. Chandra, N. Zeldovich, C. Sapuntzakis, and M. S. Lam. The collective: A cache-based system management architecture. In *In Proc. 2nd Symposium on Networked Systems Design and Implementation (NSDI*, pages 259–272, 2005.

[4] A. Clements, I. Ahmad, M. Vilayannur, and J. Li. Decentralized deduplication in SAN cluster file systems. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, page 8. USENIX Association, 2009.

[5] F. Dogar, A. Phanishayee, H. Pucha, O. Ruwase, and D. Andersen. Ditto - a system for opportunistic caching in multi-hop wireless mesh networks.

In *Proceedings of ACM MobiCom*, San Francisco, CA, Sept. 2008.

[6] 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 *FAST '09: Proceedings of the 7th Conference on File and Storage Technologies*, pages 197–210, Berkeley, CA, USA, 2009. USENIX Association.

[7] A. Gulati, C. Kumar, and I. Ahmad. Storage workload characterization and consolidation in virtualized environments. In *2nd International Workshop on Virtualization Performance: Analysis, Characterization, and Tools (VPACT)*, 2009.

[8] J. Hansen and E. Jul. Lithium: virtual machine storage for the cloud. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 15–26. ACM, 2010.

[9] X. He. A caching strategy to improve iSCSI performance. *27th Annual IEEE Conference on Local Computer Networks, 2002. Proceedings. LCN 2002.*, pages 278–285, 2002.

[10] J. Howard, M. Kazar, S. Menees, and DA. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):51–81, 1988.

[11] D. Howells and R. Ltd. Fs-cache: A network filesystem caching facility. In *Proceedings of the Linux Symposium*, volume 1, 2006.

[12] J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):3–25, 1992.

[13] H. Lagar-Cavilla, J. Whitney, A. Scannell, P. Patchin, S. Rumble, E. De Lara, M. Brudno, and M. Satyanarayanan. SnowFlock: rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 1–12. ACM, 2009.

[14] Microsoft. Virtual hard disk image format specification, Feb. 2009. `http://technet.microsoft.com/en-us/virtualserver/bb676673.aspx`.

[15] D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. Rowstron. Everest: Scaling down peak loads through I/O off-loading. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 15–28. USENIX Association, 2008.

[16] R. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman, and S. Owara. SnapMirror: file-system-based asynchronous mirroring for disaster recovery. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, page 9. USENIX Association, 2002.

[17] C. Pettey and H. Stevens. Gartner says worldwide hosted virtual desktop market to surpass $65 billion in 2013, Mar. 2009. `http://www.gartner.com/it/page.jsp?id=920814`.

[18] B. Pfaff, T. Garfinkel, and M. Rosenblum. Virtualization aware file systems: Getting beyond the limitations of virtual disks. In *In 3rd Symposium of Networked Systems Design and Implementation (NSDI*, pages 353–366, 2006.

[19] M. Satyanarayanan, J. H. Howard, D. a. Nichols, R. N. Sidebotham, A. Z. Spector, and M. J. West. The ITC distributed file system. *ACM SIGOPS Operating Systems Review*, 19(5):35–50, Dec. 1985.

[20] E. Scholten. How to: Optimize guests for vmware view, July 2010. `http://www.vmguru.nl/wordpress/2010/07/how-to-optimize-guests-for-vmware-view`.

[21] M. D. Schroeder, D. K. Gifford, and R. M. Needham. A caching file system for a programmer's workstation. In *SOSP*, pages 25–34, 1985.

[22] VMWare. Virtual machine disk format (vmdk), 2010. `http://www.vmware.com/technical-resources/interfaces/vmdk.html`.

[23] W. Vogels. File system usage in windows nt 4.0. In *ACM Symposium on Operating System Principles*, pages 93–109. ACM, 1999.

[24] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–14. USENIX Association, 2008.

Exploiting Half-wits: Smarter Storage for Low-Power Devices

Mastooreh Salajegheh*, Yue Wang†, Kevin Fu*, Anxiao (Andrew) Jiang†, Erik Learned-Miller*
**Department of Computer Science, University of Massachusetts Amherst*
†*Department of Computer Science and Engineering, Texas A&M University*
{*negin,kevinfu,elm*}*@cs.umass.edu,* {*yuewang,ajiang*}*@cse.tamu.edu*

Abstract

This work analyzes the stochastic behavior of writing to embedded flash memory at voltages lower than recommended by a microcontroller's specifications to reduce energy consumption. Flash memory integrated *within* a microcontroller typically requires the entire chip to operate on common supply voltage almost double what the CPU portion requires. Our approach tolerates a lower supply voltage so that the CPU may operate in a more energy efficient manner. Energy efficient coding algorithms then cope with flash memory that behaves unpredictably.

Our software-only coding algorithms (*in-place writes*, *multiple-place writes*, *RS-Berger codes*) enable reliable storage at low voltages on unmodified hardware by exploiting the electrically cumulative nature of half-written data in write-once bits. For a sensor monitoring application using the MSP430, coding with in-place writes reduces the overall energy consumption by 34%. In-place writes are competitive when the time spent on computation is at least four times greater than the time spent on writes to flash memory. Our evaluation shows that tightly maintaining the digital abstraction for storage in embedded flash memory comes at a significant cost to energy consumption with minimal gain in reliability.

1 Introduction

Billions of microcontrollers appear in embedded systems ranging from thermostats and utility meters to tollway payment transponders and pacemakers[1]. Recent years have witnessed a proliferation of low-power embedded devices [2, 7, 17, 21], many of which use on-chip flash memory for storage.

While the reliability, low cost, and high storage density of flash memory make it a natural choice for embedded systems [15], its relatively *high voltage requirement* (Table 1) introduces challenges for energy-efficient designs aiming to maximize the system's effective lifetime (e.g., the run time on a typical battery whose voltage declines over time). Instrumenting the system to operate at a fixed low voltage v_l is one way to reduce power consumption; however, achieving *consistently correct* results for flash writes are guaranteed only if v_l is higher than a manufacturer-specified threshold. Moreover, in energy-limited devices that cannot provide a constant supply voltage, scenarios may arise in which the flash memory is the only part of the circuit whose operating requirements are not met. In such cases, applications can expect normal operation when they are not performing flash writes and unpredictable behavior when they are.

Microcontroller	CPU Min. voltage	Flash write Min. voltage
TI MSP430 [36]	1.8 V	2.2 or 2.7 V
PIC32M [24]	2.3 V	3.0 V
ATmega128L [3]	2.7 V	4.5 V

Table 1: Flash memory restricts choices for the CPU voltage supply on microcontrollers because the CPU shares the same power rail as the on-chip flash memory.

Because embedded flash memory typically shares a common voltage supply with the CPU (separate power rails are cost prohibitive), a single voltage must be chosen that satisfies different components with different minimum voltage requirements. Current embedded systems address the voltage limitations of flash memory in one of the following ways:

i) A system can choose a high supply voltage sufficient for both reliable writes to flash memory and reliable CPU operation. This is a common choice for embedded systems with on-chip flash memory, but causes the CPU to consume more energy than necessary. For example, the TI MSP430F2131 microcontroller [36] in active mode consumes almost double the power when operat-

[1] A single manufacturer claims to have shipped over 8 billion microcontrollers http://www.microchip.com/sec/annual/FY10/.

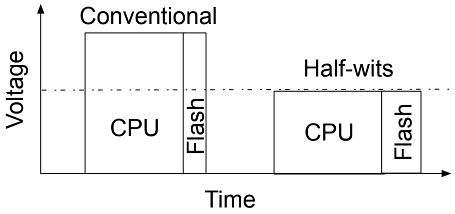


Figure 1: Operating at a lower voltage and tolerating errors instead of the conventional case of choosing the highest minimum voltage requirement may help decrease energy consumption. Considering that $Energy = voltage^2 \times time/resistance$, decreasing voltage decreases the energy consumption quadratically.

ing at 2.2 V instead of 1.8 V. Its onboard flash memory requires 2.2 V for reliable writes to flash memory.

ii) A system can choose a low supply voltage sufficient for CPU operation, but insufficient for reliable writes to flash memory. This choice allows the energy source to last longer and for the CPU to compute more efficiently. An example of such a system is the Intel WISP [33], a batteryless RFID tag that sets its operating voltage to 1.8 V—below its onboard flash memory's 2.2 V specified minimum—to save power. Flash memory cannot be written on this device. The microcontroller could use a low-power wireless interface (e.g., RF backscatter) to store data remotely. Such an approach, however, raises privacy as well as performance concerns [32].

iii) A system can modify hardware to enable dynamic voltage scaling. This approach requires additional analog circuitry such as voltage regulators and GPIO-controlled switches. Because many embedded systems are extremely cost sensitive, this choice is unattractive for high-volume manufacturing with low per-unit profit margins. An additional 50 cent part on a thermostat control can be cost prohibitive. Moreover, small changes may necessitate a new PCB layout—upsetting the delicate supply chain and invalidating stocked inventories of already fabricated PCBs.

Approach. Our approach reduces the operating voltage of the microcontroller to a point at which the resulting energy savings of the CPU portion of the workload exceeds the energy cost of the algorithms for ensuring reliable writes (Figure 1). The technique requires minimal or no hardware modification and also allows for RFID-scale devices to better exploit capacitors as power supplies. The capacitor provides finite energy and therefore the voltage decays exponentially. The long tail of the curve provides insufficient voltage for conventional writes to flash memory, but is sufficient for reliable storage with our techniques.

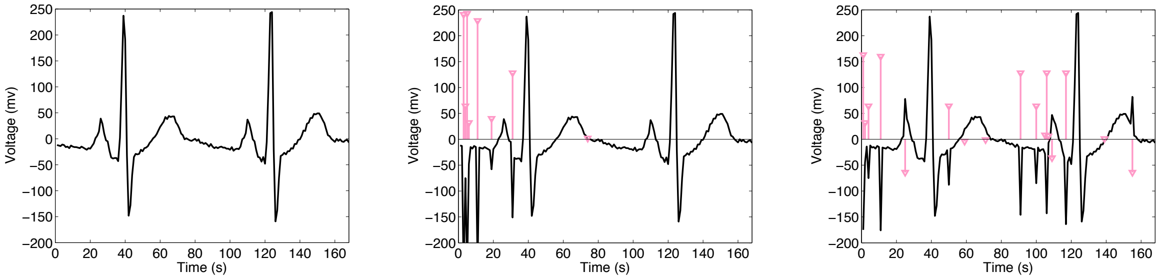
Of wits and half-wits. In 1982, Rivest and Shamir introduced the notion of write-once bits (wits) in the context of coding theory to make write-once storage behave like read-write storage [31]. Bits in flash memory behave like wits because a programmed bit cannot be reprogrammed without calling an energy-intensive erase operation to a block of memory much larger than a single write. We coin the term *half-wits* to refer to wits used in a manner inconsistent with a manufacturer's specifications, resulting in stochastic behavior. Half-wits in this work are wits of flash memory used below the recommended supply voltage.

In examining error rates at low voltage and constructing a system that provides reliable storage despite errors, our work suggests that it is appropriate to relax previously assumed constraints and reexamine the costly digital abstractions layered above on-chip flash memory.

Contributions. Our primary contributions include an empirical evaluation that characterizes the behavior of on-chip flash memory at voltages below minimum levels specified by manufacturers, and algorithms that enable reliable writes to flash memory while coping with low voltage. Our evaluation identifies three key factors affecting error rates: voltage, Hamming weight of the data, and the wear-out history of the flash memory.

The first algorithm, *in-place writes*, makes attempts at *write time* to store a value correctly in the given memory address. The *in-place writes* method repeatedly writes data to the same memory address. The intuition behind this approach is that repeating a write attempt in a consistent location accumulates the charge in the same cell, increasing the chance of storing a bit of information correctly. In addition, since flash writes only change bits in a single direction, a correctly written bit cannot be reversed to produce an error on a second write attempt. The second algorithm, *multiple-place writes*, tries to decrease the probability of error by making attempts at both write time and read time. This method stores data in more than one location aiming that the data (even partially) will be stored correctly in at least one of these locations. The third algorithm is a hybrid error-correcting code combining Reed-Solomon (RS) [29] and Berger [5] codes. The Berger code detects, but does not correct, asymmetric errors caused by the low write voltage. Given the approximate locations of errors, which are determined by the Berger code, the RS code efficiently recovers the originally stored data.

The paper compares all three methods in terms of energy consumption, execution time, and error correction rate. We also show that our methods are most effective for CPU-bound workloads. With respect to cost and energy, our techniques may enable already deployed embedded flash memory to remain competitive with emerging technology for low-power, non-volatile memory.



(a) Writes at 2.0 V

(b) Writes at 1.9 V

(c) Writes at 1.8 V

Figure 2: As operating voltage decreases, flash-write errors increase. (a) shows an original ECG signal correctly stored at 2.0 V (despite operating below the recommended threshold). As the voltage decreases in (b) and further in (c), erroneous writes (light-colored spikes, height varying according to the magnitude of the error) become more common. The black line shows the reconstructed signal that includes the errors.

2 Behavior of Storage on *Half-wits*

Before we can design effective coding algorithms, we must first understand the behavior of errors in half-wits. By tolerating a lower voltage, an energy-limited embedded device can decrease its power consumption and therefore extend its lifetime on a finite energy supply[2]. The minimum operating voltage of embedded devices that use nonvolatile on-chip storage is usually determined by the requirements of flash memory. For example, the TI MSP430 microcontroller can operate at 1.8 V, but its nominal minimum voltage for flash writing and erasure is 2.2 V (Table 1). Increasing operating voltage from 1.8 V to 2.2 V causes the CPU to draw about 50% more power without commensurate gain in clock speed because of the voltage squaring effect.

The drawback of lowering voltage below flash memory requirements in order to save power is the loss of flash memory reliability. Figure 2 shows the result of running a MSP430F2131 at three different voltages—all lower than the nominal minimum for flash writes—to store electrocardiogram (ECG) data samples from the PhysioNet database [13] in flash memory. Many medical sensor networks [20, 22, 34] that provide ECG measurements are energy limited and use on-chip flash memory as primary storage.

These graphs support the intuition that flash writes may not be error free at low voltages and that there exist voltage levels below the minimum recommended voltage at which flash writes function correctly[3]. To investigate the behavior of flash memory at low voltage and determine the factors influencing the error rate, we performed experiments on an automated testbed of our own design.

2.1 Experimental Methodology

We use a consistent experimental setup for all of the experiments in this work. Our choice of test platform is a TI MSP430 [36] microcontroller with on-chip flash memory. More specifically, we tested two types of TI microcontrollers: MSP430F2131 and MSP430F1232. The MSP430 is common in low-power embedded applications; we note especially its use in sensor motes [28] and RFID-scale batteryless devices [33]. In our setup, an MSP430 microcontroller runs a test program that involves both computation and flash operation. We power the microcontroller with an external power supply held steady at a voltage below the nominal minimum for flash writes. An external chip captures the contents of flash memory after each experiment.

To automate the testing of flash write behavior, we have developed a flash memory testbed. The two major components of the testbed are a test platform and a connected monitoring platform. The monitoring platform is based on an additional MSP430 microcontroller. The test platform runs a test program at low voltage. When the test program completes, the test platform sends the result of the experiment to the monitoring chip via GPIO pins. The test and monitoring platforms share 8+1 GPIO pins to carry one byte of data and a clock signal. Once the test platform puts data on its eight data pins, it raises the clock pin. The monitoring chip reads data from its GPIO pins whenever it detects a rising clock signal and logs the results in its own flash memory. The monitoring chip runs at a voltage above the nominal minimum for its own flash writes, and therefore stores reliably.

2.2 Unreliable, Low-Voltage Flash Memory Writes

The TI MSP430 datasheet [36] states that flash writes at any voltage lower than the nominal minimum voltage (which is 2.2 V in the case of MSP430F2131) are not guaranteed to succeed. However, as the graphs in

[2] Or because of relaxed requirements, eliminate the need for multiple batteries in series to achieve a high voltage.

[3] Moreover, a nonzero error rate may be tolerable by some applications. In the case of ECG data, the cardiac pulse interval can be recovered from noisy data stored at low voltage.

Figure 2 show, not all flash writes fail at low voltages. On the contrary, in this specific experiment, most of the writes (95.24% at 1.9 V and 89.88% at 1.8 V) succeed.

In a NOR flash memory, all cells are initialized to 1, and writing data to a byte of flash memory means setting an appropriate number of bits to 0 by applying electrical charge to the corresponding flash cells. At low voltage, there may be insufficient charge to effect a transition to 0, and a flash write may store fewer 0 bits than requested [27]. To be specific, we define errors as follows: when a byte of data d_1 is written in a flash memory address and then data d_2 is read from that address, there is an error if $d_1 \neq d_2$. An experiment, explained next, investigates the behavior of low-voltage flash memory and gives bit-level results.

Using the automated flash testbed explained in Section 2.1, the test platform runs a program that writes numbers $\{0, \cdots, 255\}$ to flash memory, then sends the contents of its flash memory to the monitoring platform via GPIO pins. Table 2 compares the written data and the intended data for cases in which errors occurred. It demonstrates that, when both are represented as integers, the absolute value of the stored data is always greater than or equal to the absolute value of the intended data.

2.3 Determining Factors That Affect Error Rates

We consider the following potential factors that may affect the error rate of setting a bit to 0 in a flash memory at low voltage: voltage level, Hamming weight of the data, wear-out history, permutation of 0s, and neighbor cells. The effects of each of these variables are evaluated by designing an experiment to test a hypothesis. All the experiments are performed on flash memories with minimal previous usage unless stated otherwise.

Voltage level: Our hypothesis is that the lower a chip's operating voltage (and that of its on-chip flash memory), the higher the error rate of flash writes. Figure 3 confirms this hypothesis; moreover, the graph shows that for different chips of exactly the same type, the error rate can be different even under equivalent voltage.

Experiment: Two MSP430F2131 and two MSP430F1232 microcontrollers run a program that writes zeros to the data segment of their flash memory. We increased the microcontroller's operating voltage in 10 mV steps, and used the monitoring platform to compute the byte error rates over 50 runs.

Hamming weight: In an erased (i.e., having value 1) flash cell, writing a 1 is always error free because no change to the cell is necessary. However, setting a cell to 0 might fail if there is not enough charge accumulated in that cell. Our hypothesis is that, the lower the Hamming weight (number of 1s in the binary representation) of a number, the higher the probability of error when writing that number to flash at low voltage.

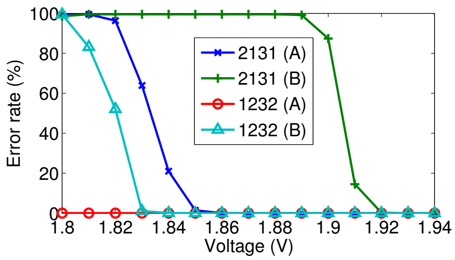


Figure 3: Flash write error rates decrease as voltage increases. This trend holds for all the chips (MSP430F2131 and MSP430F1232) we tested, though error rates differ even between chips of the same model.

Based on per-byte Hamming weight, there are nine equivalence classes of integers that can be represented in one byte. The weight-8 equivalence class has only one member, 255, which can always be written to an erased flash cell without error. The other extreme case is the weight-0 equivalence class, containing only 0s, that requires all eight bits to transition to 0. Figure 4 shows the byte error rate for all nine equivalence classes, measured via the following experiment.

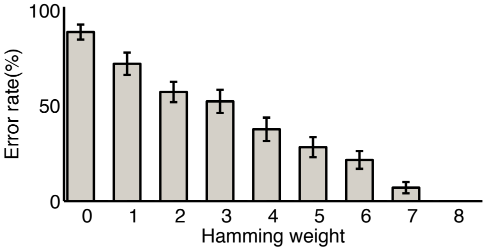


Figure 4: As the Hamming weight (number of 1s in the binary representation) of a number increases, the error rate of low-voltage flash write declines. The data corresponds to a MSP430F2131 running at 1.84 V.

Experiment: At 1.84 V, a MSP430F2131 runs a program that writes numbers from the same equivalence class to one block (64 bytes) of flash memory. We used the monitoring platform to compute the average byte error rate of flash writes for each of the nine equivalence classes over 50 runs.

Corollary: To exploit the fact that the Hamming weight of a number affects probability of error when it is written to flash, one can transform numbers into numbers with greater Hamming weights before writing them to flash memory.

Wear-out history: Flash memory has a limited lifetime (about 10^5 cycles of erasures) after which the erase operations fail to reliably reset the bits to 1. We suspect that the more flash memory is erased (worn-out), the

(Binary)	Intended	00001100	00001101	00001110	00010100	00100111	10100100
	Written	11101101	01011111	11111111	11111111	00101111	10101111
Hamming distance		4	3	5	6	1	3

Table 2: Erroneous flash writes at low voltage. Insufficient electrical charge may result in some bits failing to transition from 1 (the initial state) to 0.

lower its error rate of setting bits to 0 would become[4]. Figure 5 shows a heat map of bit error rate for three blocks of flash memory (192 bytes) on an MSP430F2131 microprocessor. Lighter colors in the heat map represent higher error rates. The disproportionately dark color of the middle block is due to more frequent erasure of that block compared to the other two blocks.

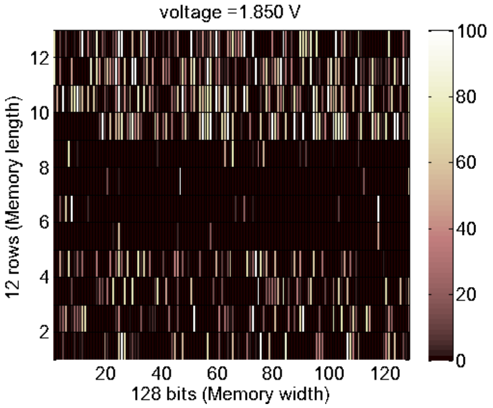


Figure 5: Worn-out flash memory blocks are biased toward ease of writing zeros. Lighter color represents higher average number of errors over 50 trials. The middle block has been write/erase cycled 6,000 times. The other two blocks are minimally used.

Experiment: A MSP430F2131 runs a program that writes zeros to all three blocks of its flash memory. The MSP430 is first worn out such that one block has 6,000 write/erase cycles and two blocks have minimal previous usage. We used the monitoring platform to compute the average error rate for all bits in the three blocks of memory over 50 trials.

Corollary: Wear-out history affects error rate, so storing data in more than one location might help decrease the error rate, especially if those locations are in different blocks of memory.

Permutation of 0s: Two numbers belonging to the same Hamming-weight equivalence class can have different permutations of 0 bits. We tested to see if the error rate depends on the permutation of 0s in one byte of data. For example, the numbers 240, 15, 170, and 71 all have four 0s in their binary representation but in different places (240 has 0s in the right nibble, and 15 has all of its 0s in its left nibble, etc.). The result of the experiment shows a similar byte error rate with mean of 39.85 ± 4.29% for numbers in the same equivalence class. The small standard deviation (4.29%) shows that the permutation of 0s does not significantly affect the error rate and therefore we do not consider this factor in our design directions.

Experiment: A MSP430F2131 runs a program that cycles through eight numbers from the same Hamming-weight equivalence class, writing them to 192 consecutive bytes of flash memory. We used the monitoring platform to compute the average error rates for each of the 192 bytes over 50 trials.

Neighbor cells: Another factor that might affect the error rate of storage in a flash cell at low voltage is the values of neighboring cells. However, our results suggest that a cell's error rate does not appear to depend on the values stored in neighboring cells (Figure 6).

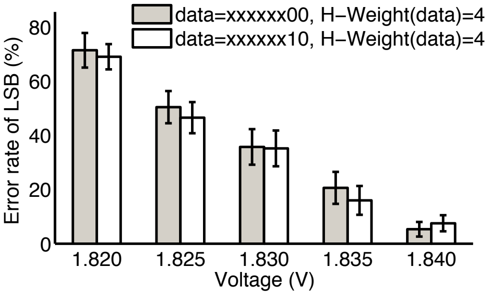


Figure 6: Error rate of a cell is not noticeably influenced by the value of its neighbor. The graph shows that the value of the second LSB does not greatly affect the error rate of the LSB. The bars show the error rate of the LSB for writing numbers from the same Hamming-weight equivalence class whose two LSBs are set to either 00 (dark bars) or to 10 (light bars).

Experiment: In order to determine if the error rate of a cell is affected by its neighbor, we consider all numbers from the same Hamming-weight equivalence class whose two Least Significant Bits (LSBs) are set to either 00 (case 1) or 10 (case 2). An example of case 1 is number 60 (0b00111100) and an example of case 2 is number 30 (0b00011110). This experiment fixes the Hamming weight variable and changes the neighbor value of the

[4] This counterintuitive hypothesis is consistent with the notion that flash erasures (settings bits to 1) become harder with wear out.

LSB to be 0 or 1. We deem a write erroneous if the LSB is not set to 0. The experiment was done for a Hamming weight of four and it was repeated for five voltage levels in the interval of 1.82 V to 1.84 V with steps of 5 mV. The error rate for any voltage above 1.84 V was close to 0% and for any voltage below 1.82 was close to 100%. We used the monitoring platform to compute the average error rates of case 1 and case 2 for each of the voltage levels over 50 trials.

2.4 Accumulative Memory Behavior

It is helpful to understand a few details of the electrical nature of flash memory in order to appreciate the expected behavior of conventional digital abstractions when layered above embedded flash memory. Each flash memory cell is a floating-gate (FG) transistor made up of a source, drain, control gate, and floating gate. The floating gate is separated from the source and drain by an insulating oxide layer that makes it difficult for electrons to travel into or out of the gate. Flash cells rely on this oxide to maintain logical state in the absence of power, making the memory non-volatile [27].

To write a memory cell (which has an erased value of 1), the control circuitry applies a high field to the source. The application of this field greatly increases the probability that electrons in the floating gate will tunnel to the source. If a sufficient number of electrons tunnel to the source, the cell is subsequently read as a 0. To erase a cell (restoring a 1), the control circuitry applies a high field to both the source and drain. This field energizes the electrons currently stored near the source, allowing them to jump the oxide barrier to the floating gate where they are once again trapped [27].

Not all electrons must transition in order for a write or erase operation to be successful. The operation only needs to change the state of some majority of the electrons so that subsequent read operations detect sufficient charge to discern the intended value. Lowering the applied voltage (and thus the field strength) lowers the probability of state change for each electron but, as noted earlier, electrons that do transition will remain in place.

A low-power storage scheme can benefit from this accumulative property by repeating writes to the same cell. Each write operation will increase the chance of success by forcing some number of state transitions. In other words, a failed write is still progress.

3 Design of a Low-Voltage Storage

This section presents our design for a software system that enables reliable flash memory writes at low voltage. We first present a model that captures the basic characteristics and behavior of flash memory. We then set design goals with that model under consideration. We introduce three methods for reliable flash storage, which we refer to as *in-place writes*, *multiple-place writes*, and *RS-Berger codes*. Each method aims to meet our design goals for reliable non-volatile storage.

3.1 Modeling Low-Voltage Flash Memory

A NOR flash memory has a set of n cells that are initially set to 1. We represent the state of the cells by $c_1, \ldots, c_n$; the value of c_i can be 0 or 1. A cell can be set to 0 using a write operation. The $1 \rightarrow 0$ transition might fail at low voltage while the $1 \rightarrow 1$ will obviously succeed. Flash memory at low voltage, where errors occur only in one direction, can be modeled as a Z-channel [19].

Flash memory is a write-once memory [31] and once a cell is set to 0 (i.e., once it is programmed), it cannot be changed back to 1 without using an erase operation. In flash memory, cells are organized by blocks, and an erase operation resets an entire block of cells. Block erasures are costly in terms of time and energy and they cause wear to flash cells.

Operations: There are two operations in this model: (1) An update operation that changes a subset of cells to 0 to represent a value, and (2) A decoding operation that maps cell states (i.e., memory state) to a value. Updating a variable means changing the values of $c_1, \ldots, c_n$ to $c'_1, \ldots, c'_n$. Assuming no erase operation occurs, and therefore no bits are reset to 1 after being set to 0, we have $\forall i \in \{1, \ldots, n\}, c_i \geq c'_i$ after an update. If the update operation is performed when operating voltage is below the nominal minimum required for flash memory, the update operation may not be error free.

3.2 Design Goals

Our storage techniques, which aim to provide reliable storage for low-power devices, are designed with the following metrics in mind:

- *Error rate:* The first and foremost design goal is to minimize the error rate to provide applications with reliable non-volatile storage.
- *Energy consumption:* The energy consumed to achieve an acceptably low error rate should not exceed the expected energy savings gained by running at a lower voltage.
- *Delay:* We define delay as the difference between the execution time to reliably store data at a low voltage and to store the same data at a high voltage. The delay caused by the storage technique should be reasonably small.

3.3 Proposed Methods

Toward the design goals discussed previously, we propose methods to deal with errors caused by using flash memory at low voltage.

3.3.1 In-Place Writes

Since the transition of a 1 to a 0 in a NOR flash memory at low voltage is stochastic rather than guaranteed, the *in-place writes* method repeats the write of each byte (to the same memory location) more than once if necessary, up to a *threshold* number of attempts. Algorithm 1 gives the details for ENCODE and DECODE procedures for in-place writes.

Algorithm 1 The encoding and decoding algorithms for *in-place writes* method to store *data* to *address* by repeating the writes up to *threshold* a number of attempts if necessary.

```
ENCODE($data, address, threshold$)
WRITE_TO_FLASH($data$,$address$)
$result \leftarrow$ READ_FROM_FLASH($address$)
$repeat \leftarrow 1$
while ($result \neq data$) AND ($repeat < threshold$)
    do WRITE_TO_FLASH($data$,$address$)
       $result \leftarrow$ READ_FROM_FLASH($address$)
       $repeat \leftarrow repeat + 1$


DECODE($address$)
$result \leftarrow$ READ_FROM_FLASH($address$)
return $result$
```

The reason *in-place writes* decrease the error rate is that, as explained in Section 2.4, each write attempt in the same memory location increases the accumulated charge and therefore raises the probability of storing the intended bit sequence successfully.

3.3.2 Multiple-Place Writes

Another approach to increase the reliability of flash writes at low voltage is to write a value to more than one location in flash memory if necessary up to a *threshold* number of locations. Later, to retrieve the stored data, the *multiple-place writes* method reads the data from the specified address and several other addresses associated with it, then returns the bitwise AND of all of the stored values. Algorithm 2 details ENCODE and DECODE procedures of the *multiple-place writes* method. Writing a value to more than one memory location increases the probability of storing it successfully in the flash memory.

The reason the *multiple-place writes* approach can decrease the error rate is as follows: All cells of flash memory are initially set to 1. An error means that writing a 0 has failed and a bit cell c_i has remained untouched (logical 1) although it was intended to be set to 0. If the cell write in one of the locations has not failed, and cell c_i is 0 in at least one location, getting the AND of the read values from all locations will make cell $c_i = 0$ in the AND result. The case of writing a 1 to a cell does not cause an error since it means changing a cell from 1 to 1.

Algorithm 2 The encoding and decoding algorithms for *multiple-place writes* method to store *data* to *address* by repeating the writes up to a *threshold* number of locations if necessary. The distance between each of these associated locations is *offset*.

```
ENCODE($data, addr, threshold, offset$)
WRITE_TO_FLASH($data$,$addr$)
$result \leftarrow$ READ_FROM_FLASH($addr$)
$repeat \leftarrow 1$
while ($result \neq data$) and ($repeat < threshold$)
    do $phy\_addr \leftarrow addr$ + ($repeat \times offset$)
       WRITE_TO_FLASH($data$,$phy\_addr$)
       $n\_result \leftarrow$ READ_FROM_FLASH($phy\_addr$)
       $result \leftarrow result\ \&\ n\_result$
       $repeat \leftarrow repeat + 1$


DECODE($addr, threshold, offset$)
for $i \leftarrow 0$ to $(threshold - 1)$
    do $phy \leftarrow addr$ + ($i \times offset$)
       $n\_result \leftarrow$ READ_FROM_FLASH($phy$)
       $result \leftarrow result\ \&\ n\_result$
return $result$
```

3.3.3 RS-Berger Codes

Our third method to provide reliable flash memory at low voltage involves data coding. We use the concatenation of Reed-Solomon [29] and Berger [29] codes—which we call *RS-Berger codes*—to detect and correct errors at read time. Reed-Solomon is a widely used error-correcting code that can correct twice as many erasures as errors. Therefore, if the locations of errors are known, an RS code's error-correcting capacity is improved twofold.

To detect the location of errors and therefore improve the efficiency of the RS code, we use a Berger code, an error-detecting code for asymmetric channels. As previously mentioned (Section 3.1), flash memory at low voltage can be modeled as a Z-channel for which a Berger code is suitable. A Berger codeword consists of two parts: k information bits and $\lceil \log_2(k+1) \rceil$ check bits. The check bits of the Berger codeword represents the number of zeros in the k information bits. The Berger code can detect all zero-to-one errors, because the number of zeros in the information-bit component will always be less than the number represented by the check-bit component.

To represent RS-Berger codewords, we use a matrix in which each row is an RS codeword except for the last row which includes the Berger check bits of the RS codewords. In other words, each cell in the last row of the matrix is the sum of the number of zeros in the corresponding cells in all other rows.

When encoding the data, we first use RS code to generate *n* codewords (rows of the matrix) and then we apply a Berger code to compute the check bits for each symbol for all codewords (each column of the matrix).

When decoding data, we first use the Berger decoder to check whether or not each column is erroneous. If one entry in the column is erroneous, we consider all the symbols in the column erasures; otherwise, all the symbols in the column are considered correct. Then, once the error locations are known, we apply RS decoding to correct the erroneous sequences row by row.

Algorithm 3 The encoding and decoding algorithms for *RS-Berger codes* write method. *t* is the maximum number of errors RS-Berger code can correct.

```
ENCODE($data_{1,..,N}$, $n$)
for $i \leftarrow 1$ to $N$
    do $CW_i \leftarrow$ RS_ENCODE($data_i$, $n$)
        WRITE_TO_FLASH($CW_i$, $address_i$)
for $i \leftarrow 1$ to $n$
    do for $j \leftarrow 1$ to $N$
            do $sym_{i,j} \leftarrow CW_j$(i)
        $chk_i \leftarrow$ BERGER_ENCODE($sym_{i,(1,..,N)}$)
        WRITE_TO_FLASH($chk_i$, $address_{N+1} + i\text{-}1$)
```

```
DECODE($addr_{1,..,(N+1)}$, $n$, $t$)
for $i \leftarrow 1$ to $N$
    do $chk_i \leftarrow$ READ_FROM_FLASH($addr_{N+1}$+$i$-$1$)
for $i \leftarrow 1$ to $N$
    do $CW_i \leftarrow$ READ_FROM_FLASH($addr_i$)
        for $j \leftarrow 1$ to $n$
            do $sym_{i,j} \leftarrow CW_i(j)$
$errors \leftarrow \{\}$
for $i \leftarrow 1$ to $n$
    do $err \leftarrow$ BERGER_DECODE($sym_{i,(1,..,N)}$, $chk_i$)
        if $err = 0$
            then $errors \leftarrow errors \cup \{i\}$
if $|errors| \leq t$
    then for $i \leftarrow 1$ to N
            do $result_i \leftarrow$ RS_DECODE($CW_i$, $errors$)
        return $result$
    else return “fail to correct errors”
```

4 Evaluation

Our storage techniques are designed for the resource limitations of low-power devices. In this section, we first evaluate the suitability of the three methods proposed in Section 3.3 for low-power devices; we then evaluate the hypothesis that for CPU-bound workloads, operating at low voltage and managing errors is more energy efficient than fixing the operating voltage to the maximum of all the components' nominal minimum voltages.

Summary of results: For a sensor monitoring application that reads 256 data samples from flash memory, aggregates data, and stores the results in flash memory, use of *in-place writes* at 1.8 V reduces the energy consumption up to 34% versus running the same application at 2.2 V (minimum voltage requirement for the flash memory). This sensing application models a common workload for both wireless sensor nodes and RFID-scale devices.

Experimental setup: We used a consistent experimental setup to measure the energy consumption and execution time of each program. Using an oscilloscope, we measured the voltage of a small resistor in series with a MSP430 microcontroller programmed with a task (e.g., a flash write). The integration of the current (voltage divided by the resistance) over the execution time of the task multiplied by the operating voltage of the device gives the energy consumption of that task ($Energy = \int I(t)\ dt \times V$). To facilitate precise identification of the task on the oscilloscope, the microcontroller toggled a GPIO pin immediately before and after the task.

4.1 Comparison of the Proposed Storage Methods

The workload used to measure the performance of each of the proposed methods is the storage of accelerometer traces—generated using the Intel WISP 4.1's 10-bit ADC sensor—to flash memory. The input trace is a series of three-dimensional 16-bit samples containing ten bits of information. We used a simple data compression method to store more data in the available flash memory. The compression method involved reading four samples of data, preparing the first byte of each sample to be stored in flash memory, then combining the remaining two bits of each sample into one byte of data. Using this compression scheme, we reduced every four samples (eight bytes) to five bytes.

The maximum number of write attempts for both *in-place writes* and *multiple-place* methods were set to two. The *RS-Berger codes* used three codewords of size 38 bytes (32 bytes data and 6 bytes parity). These settings enable all three methods to fit their data in 192 bytes of flash memory. Table 3 shows the energy consumption and time taken for the same workload under each method. Both *in-place writes* and *multiple-place writes* consume less energy and finish more quickly at 1.9 V than at 1.8 V. Both of these methods are feedback based and repeat writes if they detect errors. Because there is a lower chance of error at 1.9 V, fewer rewrites are required than at 1.8 V, so less energy and time are required.

The *in-place writes* method slightly outperforms the *multiple-place writes* method at both voltage levels because its decoding procedure is less CPU intensive. *In-place writes* method has the best Error Correction Rate (ECR in Table 3) of all. The *multiple-place writes* method seems to be the most suitable when there are some memory cells that are hard to program and therefore rewriting in those cells is not helpful (Figure 5 gives an example of such case). Compared to *RS-Berger codes* that always guarantee that a certain number of errors can be corrected, the *in-place writes* and *multiple-place writes* methods are less reliable—they offer no such guarantees. Therefore, for applications with a hard reliability requirement, *RS-Berger codes* may be more suitable if the application knows the error rate in advance and is willing to incur extra computational costs for RS-Berger encoding and decoding.

Method	V	Time (ms)	E (μJ)	ECR
In-place	1.8	24.16	59	96%
M-place	1.8	25.00	63	84%
RS-B	1.8	334.45	160	0%
In-place	1.9	15.43	38	100%
M-place	1.9	16.85	40	100%
RS-B	1.9	334.73	180	100%

Table 3: Performance comparison of the proposed methods at 1.8 V and 1.9 V. Error Correction Rate (ECR) shows the effectiveness of the methods.

Error Correction Rate: As Table 3 illustrates, the two methods that do not use coding—in-place writes and multiple-place writes—incur similar energy consumption costs. We now compare the effectiveness of these two approaches with respect to the error correction rate.

Figure 7 and Figure 8 demonstrate that flash storage reliability improves as we increase the number of repeated writes/places at five different voltage levels (all below the nominal minimum voltage for flash writes).

Experiment: Using our automated testbed, the test platform runs a program that writes zeros to 192 consecutive bytes of flash memory (using *in-place writes* and *multiple-place writes* methods in two different experiments). We increase the maximum number of repeated writes from one to ten, one unit at a time. The monitoring platform counts the number of incorrectly stored bytes (those that are not set to zero after the experiment). The experiment was repeated for five different voltages (1.86 V–1.90 V).

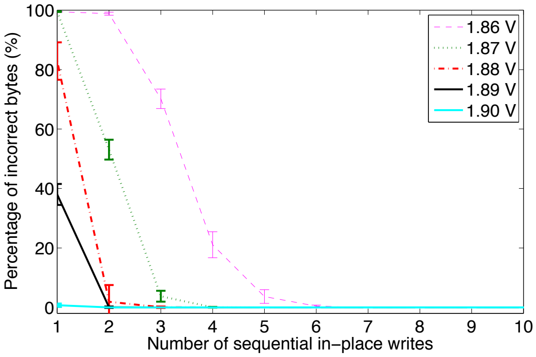


Figure 7: Reliability improvement using *in-place writes* over five different voltages.

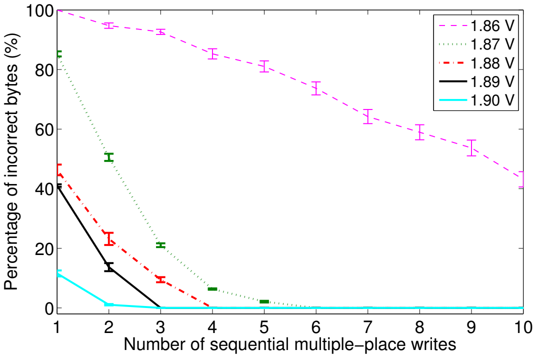


Figure 8: Reliability improvement using *multiple-place writes* over five different voltages.

Figure 9 compares the error rate of the *in-place* and *multiple-place* write methods. We choose the same maximum number of repeated writes for both approaches. As the graph shows, the *in-place writes* method improves the error rate more dramatically. We attribute this phenomenon to the fact that electrons accumulate in flash cells with each programming attempt. Figure 9 also allows us to evaluate hybrids of the *in-place writes* and *multiple-place writes* methods. For example, choosing one place to write the value and repeating the write up to three times (up to three writes in total) works better than repeating the write up to twice in two places (up to four writes in total). This graph offers evidence that a pure *in-place writes* approach works better than a hybrid approach or a pure *multiple-place writes* approach. However, we do not conclude that the *in-place writes* method always outperforms the *multiple-place writes*. A winning case for *multiple-place writes* is when a flash memory has unbalanced blocks (different error rates), for example, the chip shown in Figure 5. While *multiple-place writes* method requires more space, it could provide a more reliable storage compared to *in-place writes*.

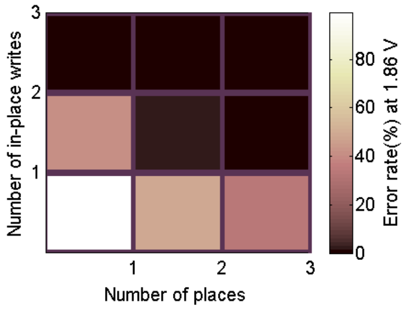


Figure 9: The *in-place writes* method reduces the error rate more effectively than *multiple-place writes* and a hybrid of both methods.

4.2 *Half-wits* Versus Wits in Practice

To evaluate our storage schemes, we consider three test cases representing CPU operations, flash read operations, and flash write operations.

The RC5 [30] test case, a CPU-only workload, is a commonly used encryption algorithm that can cope with the resource limitations of low-power devices [8, 18]. RC5 was implemented with a 32-bit word size, 18 rounds, and 16 bytes of secret key.

The retrieve and store test cases are both I/O-bound tasks. One reads and the other one writes 192 bytes of data from/to flash memory. CPU-bound operations in these test cases are minimal (essentially only a loop that calls a function to flash memory). The store program uses *in-place writes* with a maximum number of three (re)writes to deal with errors. Because flash read operations are fundamentally simpler than flash write operations, flash reads are reliable at low voltage.

We run each of the three test cases on a MSP430F2131 microcontroller at four different voltages that are all in the operating range of this microcontroller (1.8 V–3.5 V). Two voltage levels are below the recommended threshold for flash memory: 1.8 V and 1.9 V. Two voltage levels are at and above the recommended threshold: 2.2 V and 3.0 V. The microcontroller is set to work at its highest possible clock rate for each voltage level in order to gain the best energy performance. Figure 10 compares the average energy consumption over five trials of each test case at each voltage. By running at 1.8 V (below the nominal minimum voltage for flash writes on the MSP430F2131), the microcontroller consumes 48% and 33% less energy to finish the RC5 and retrieve test cases respectively. However, our storage schemes do not seem to be beneficial for flash-write-intensive tasks (the store test case).

To evaluate the end-to-end performance of our storage methods, we have tested a sensor-monitoring application that is CPU-intensive and can benefit from a low-voltage storage. This application reads from flash memory 256 accelerometer samples (each ten bits), computes the maximum, minimum, mean, and standard deviation of the samples, and stores the aggregate information in flash memory. This monitoring application is a blend of CPU and I/O, but it is still a CPU-intensive workload. Table 4 shows that providing the system with a low-voltage storage mechanism via our methods helps to decrease the task's total energy consumption by 34%.

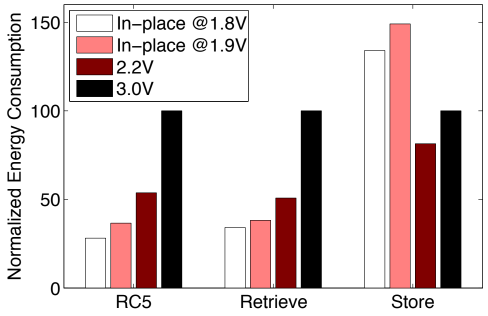


Figure 10: Micro-benchmarks: CPU (RC5), read (retrieve), and write (store) energy consumption measured at four different voltage levels. Although the RC5 and retrieve test cases consume less energy at low voltage, this is not the case for the store test case (a write-intensive application) as the savings due to running the chip at low voltage does not compensate for the energy cost required to correct errors.

4.3 Finding a Crossover Point

We can empirically find the point at which the energy saved on computation compensates for the added cost of repeated flash writes. We compare a workload executed at 2.2 V to the same one running at 1.8 V using the *in-place writes* scheme with the threshold k set to 2. We make the worst-case assumption that all data must be written to flash twice (no bits change on the first attempt). The time spent on flash writes while running at 1.8 V is

Method	In-place 1.8 V	In-place 1.9 V	None 2.2 V	None 3.0 V
Clock rate	6 MHz	6 MHz	8 MHz	14 MHz
Energy(μJ)	270	300	410	760
Time(ms)	151.15	151.32	113.24	64.72

Table 4: Energy consumption and execution time for the accelerometer sensor application. At voltages below the recommended (1.8 V and 1.9 V), *in-place writes* method with a threshold of two is used.

then twice the time spent when operating at 2.2 V. We also assume that the clock rate of the system is set to the highest specified for the CPU at each voltage. Specifically, the clock rate would be set to 6 MHz at 1.8 V and to 8 MHz at 2.2 V.

We empirically determined the power consumption of CPU and flash writes with 1.8 V and 2.2 V voltage supplies. $P_{C\_1.8} = 1.8\ mW$, $P_{C\_2.2} = 3.4\ mW$, $P_{F\_1.8} = 3.7\ mW$, and $P_{F\_2.2} = 5.8\ mW$. The variables T_C and T_F are the time spent in computation and on flash memory respectively. With these assumptions, we can write the following inequality to determine whether a given workload is likely to result in reduced energy consumption:

$$\begin{gathered} Energy_{1.8} \le Energy_{2.2} \Rightarrow \\ P_{C\_1.8} \times T_{C\_1.8} + P_{F\_1.8} \times k \times T_{F\_1.8} \le \\ P_{C\_2.2} \times T_{C\_2.2} + P_{F\_2.2} \times T_{F\_2.2} \Rightarrow \\ P_{C\_1.8} \times \frac{8MHz}{6MHz} \times T_{C\_2.2} + P_{F\_1.8} \times k \times \frac{8MHz}{6MHz} \times T_{F\_2.2} \le \\ P_{C\_2.2} \times T_{C\_2.2} + P_{F\_2.2} \times T_{F\_2.2} \end{gathered}$$

The solution with k = 2 is $T_{C\_2.2} \ge 4 \times T_{F\_2.2}$. Therefore, *in-place writes* are competitive over normal flash writes when the time spent on computation is at least four times greater than the time spent on flash writes.

5 Improvements and Alternatives

This section describes several complementary ways to further decrease the energy requirements of our schemes.

Hardware. One could add an adjustable voltage regulator and about a dozen other analog components such that software could toggle a GPIO for discrete dynamic voltage scaling. A feedback loop that dynamically adjusts a voltage supply could help identify the minimum voltage at which no write errors are detected, but such boundaries can vary with temperature and wear-out. Thus, our coding algorithms would remain helpful to cope with potential errors. Our work seeks to avoid hardware modification that would require additional components or design changes to a Printed Circuit Board (PCB) because embedded applications are often cost sensitive. Changing the PCB layout may require a manufacturer to flush its supply chain of parts typically manufactured in high volume. If an inexpensive, software-only approach with minimal disturbance to manufacturing can lead to significant savings in energy consumption, then it is hard to financially justify an expensive hardware approach that offers only comparable performance.

Sign bits and storing complements. As discussed in Section 2.3, one of the major factors influencing the error rate is the Hamming weight of a number. One way to improve the performance of the low-voltage storage methods is to store numbers with greater Hamming weights (*weight* $\ge$ 4) in flash memory. If a number is *lightweight* (*weight* $<$ 4), the complement of the number would be stored and a *sign bit* would be set for future data access. An array of sign bits can be stored separately from the data to avoid disturbing word alignment. A previous work [26] uses a similar technique for multi-level cell (MLC) flash memories with four levels; their techniques result in a significant decrease of energy consumption. Figure 11 shows that using the *sign-bit* scheme decreases the error rate at low voltage for the same ECG data used in Section 2. For this specific example, out of 168 bytes of ECG data, 160 bytes are *overweight* and therefore using the *sign-bit* scheme greatly decreased the error rate. The *sign-bit* approach involves very lightweight computation (counting the number of ones) and increases the number of writes by a factor of one-eighth. Therefore, the effect of this improvement on energy consumption and delay should be comparatively small.

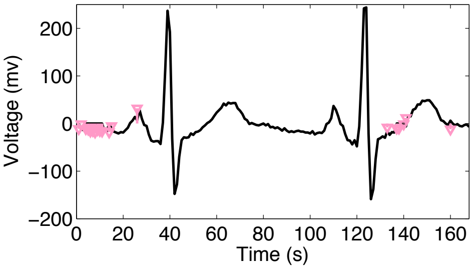


Figure 11: ECG data stored in flash memory at 1.89 V (the same chip from Figure 2) improved by using a sign bit. The light-colored bars show the difference between the ECG stored at low voltage and the original ECG data.

Memory mapping table. Another method to exploit the fact that numbers with greater Hamming weights have a lower probability of error is to map the most frequently used numbers in the user's data to the *heavier* numbers. The solution we suggest is to preprocess the data to sort numbers based on their frequency of use. A simple memory mapping table would map the most frequent numbers to the heaviest numbers. Such a table could be preloaded in flash memory so that storing the table would not consume energy at run time. Use of a memory mapping table would only increase the number of reads and would not increase the number of writes. Therefore, the energy consumption overhead and the delay should be smaller than the *sign bit* method.

An ideal, unrealizable scheme. We initially tried to set the voltage to a level lower than recommended but high enough to avoid errors. This method could not be realized for two reasons: finding a voltage that satisfies this condition requires a large number of experiments per chip—error rate varies chip by chip (Figure 3), and the error rate of flash writes varies depending on its lifespan and its environment. We found that the byte error rate of MSP430F2131 that is 63% at 1.83 V at 25°C becomes negligible when the temperature goes up to 39°C.

6 Related Work

Storage for low-power embedded devices: Recent research focuses on optimizing use of off-chip flash memory. Off-chip memory allows for special features and larger memories than found on microcontrollers, but introduces additional costs for components. Microhash [38] is a memory index structure tailored for sensor devices with a large external flash memory. Mathur et al. [23] perform an extensive study of available flash memory candidates for sensor devices and demonstrate that an off-chip parallel NAND flash memory decreases the energy consumption of storage. Considering the off-chip NAND flash memory as the best candidate for sensor devices, Agrawal et al. [1] propose a method that allows sensor devices to exploit their flash memory while adapting to different amount of RAM. However, our storage schemes are designed for already deployed low-power devices that use on-chip flash memory. Moreover, while devices at the scale of sensor nodes might switch to block-grained, large off-chip flash memory, RFID-scale platforms might not benefit from this transition because of their challenging resource limitations to drive I/O.

Energy proportionality: Our approaches share the philosophy that energy consumption should scale proportionally to utilization or error rates rather than proportional to a worst-case scenario. Blaauw et al. [6] reduce power consumption by lowering the operating voltage of a pipelined CPU. Certain pipeline stages may produce incorrect computation that require recomputation, but the errors can be made rare to allow better scalability of power consumption. Misailovic et al. [25] demonstrate that the programs whose loops performs fewer iterations cause tolerable errors while their execution time becomes shorter. Weddle et al. [37] introduce PARAID, a scheme that scales power based on the user demand while maintaining the reliability of the system. Their present work also tries to scale power based on the utilization of flash memory without losing storage reliability. Our approaches share this philosophy of scaling performance with utilization. Our performance metric is energy consumption, writes to flash memory represent our utilization, and energy-efficient error correction is our coping mechanism.

Error correction codes for storage: Most previously published flash error correction codes [9, 11, 14] are designed for NAND flash memory. Chen et al. [10] mention that NOR flash normally does not require error correction. These techniques consider neither the asymmetry in low-voltage flash memory nor the resource limitations of low-power embedded devices. Many previous codes [4, 16, 40, 35] leverage the fact that each cell of MLC flash memory represents more than one bit of information. But the fact that single-level cells (SLC) are more suitable for embedded devices, in addition to the occurrence of errors in low-voltage conditions, requires a reconsideration of these codes for SLCs at low voltage. Zemor et al. [39] introduce error-correcting WOM codes for flash memory. They suggest codes that are able to correct up to one error when the flash memory is given enough voltage. This work does not account for errors that occur at low voltage. Godard et al. [12] propose hierarchical code correction and reliability management for NOR flash memory. This work considers on-chip ECCs such as Hamming and parity codes to correct the errors in NOR flash memory.

7 Conclusions and Future Work

The high voltage requirement of on-chip flash memory is a barrier to reducing the total energy consumption of low-power devices. This work examines the main factors affecting the behavior of flash memory at low voltage. Based on our observations of flash memory behavior at low voltage, we proposed three storage schemes—*in-place writes*, *multiple-place writes*, and *RS-Berger codes*—that aim to make flash memory available and reliable at low voltage while tolerating the resource limitations of low-power devices. Our evaluation shows that in-place writes can save 34% of energy consumption for a sensing workload on the MSP430 microcontroller.

Future work includes finding more energy-efficient coding schemes to combat flash write errors caused by low voltage. Currently, the system cannot take full advantage of dynamic voltage scaling. Another plan is to introduce benchmarks for the storage systems of low-power devices. The standard benchmarks used to evaluate the storage systems designed for desktop computers are not immediately applicable to the low-power domain.

Acknowledgments

This material is supported by a Sloan Research Fellowship and the NSF under CAREER Award CCF-0747415, CNS-0627476 (prime), CNS-0627529, CAREER Award CNS-0845874, CNS-0923313, and ECCS-0802107. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

We thank Shane Clark, Wendy Cooper, Marc Liberatore, and Benjamin Ransford for feedback on drafts; Joshua Smith and Alanson Sample at Intel Labs Seattle for providing the WISP over the last three years; and our shepherd Brian Noble and the anonymous reviewers for their detailed feedback and guidance. Portions of this work are patent pending in the United States.

References

[1] D. Agrawal, B. Li, Z. Cao, D. Ganesan, Y. Diao, and P. Shenoy. Exploiting the interplay between memory and flash storage in embedded sensor devices. In *Proceedings of the 16th IEEE Conference on Embedded and Real-time Computing Systems (RTCSA)*, pages 227–236, 2010.

[2] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38(4):393–422, 2002.

[3] Atmel AVR Solutions. ATmega128L. http://www.atmel.com/atmel/acrobat/doc2467.pdf.

[4] A. Barg and A. Mazumdar. Codes in permutations and error correction for rank modulation. *IEEE Transactions on Information Theory*, 56(7):3158–3165, 2010.

[5] J. Berger. A note on error detection codes for asymmetric channels. *Information and Control*, 4(1):68–73, 1961.

[6] D. Blaauw and S. Das. CPU, heal thyself. *IEEE Spectrum*, 46(8):40–56, 2009.

[7] M. Buettner, B. Greenstein, A. Sample, J. R. Smith, and D. Wetherall. Revisiting smart dust with RFID sensor networks. In *Proceedings of the 7th ACM Workshop on Hot Topics in Networks (HotNets-VII)*, October 2008.

[8] H.-J. Chae, D. J. Yeager, J. R. Smith, and K. Fu. Maximalist cryptography and computation on the WISP UHF RFID tag. In *Proceedings of the Conference on RFID Security*, July 2007.

[9] B. Chen, X. Zhang, and Z. Wang. Error correction for multi-level NAND flash memory using Reed-Solomon codes. In *IEEE Workshop on Signal Processing Systems (SiPS 2008)*, pages 94–99, Oct. 2008.

[10] S. Chen. What types of ECC should be used on flash memory? Application Note for SPANSION, 2007.

[11] M. Fujino and V. Moshnyaga. An efficient Hamming distance comparator for low-power applications. In *9th International Conference on Electronics, Circuits and Systems*, volume 2, pages 641–644, 2002.

[12] B. Godard, J.-M. Daga, L. Torres, and G. Sassatelli. Hierarchical code correction and reliability management in embedded NOR flash memories. In *Proceedings of the 2008 13th European Test Symposium*, pages 84–90, 2008.

[13] A. L. Goldberger, L. A. N. Amaral, L. Glass, J. M. Hausdorff, P. C. Ivanov, R. G. Mark, J. E. Mietus, G. B. Moody, C.-K. Peng, and H. E. Stanley. PhysioBank, PhysioToolkit, and PhysioNet: Components of a new research resource for complex physiologic signals. *Circulation*, 101(23):e215–e220, 2000. Circulation Electronic Pages: http://circ.ahajournals.org/cgi/content/full/101/23/e215.

[14] S. Gregori, A. Cabrini, O. Khouri, and G. Torelli. On-chip error correcting techniques for new-generation flash memories. *Proceedings of the IEEE*, 91(4):602–616, April 2003.

[15] A. Jiang, R. Mateescu, M. Schwartz, and J. Bruck. Rank modulation for flash memories. In *IEEE International Symposium on Information Theory (ISIT)*, pages 1731–1735, 2008.

[16] A. Jiang, M. Schwartz, and J. Bruck. Correcting charge-constrained errors in the rank-modulation scheme. *IEEE Transactions on Information Theory*, 56(5):2112–2120, 2010.

[17] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Next century challenges: mobile networking for "Smart Dust". In *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom)*, pages 271–278, 1999.

[18] C. Karlof, N. Sastry, and D. Wagner. TinySec: A link layer security architecture for wireless sensor networks. In *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2004.

[19] T. Klove. Error correcting codes for the asymmetric channel. Technical report, Informatics, University of Bergen, 1995.

[20] B. P. L. Lo, S. Thiemjarus, R. King, and G. zhong Yang. Body sensor network - a wireless sensor platform for pervasive healthcare monitoring. In *Adjunct Proceedings of the 3rd International Conference on Pervasive Computing (PERVASIVE)*, pages 77–80, 2005.

[21] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless sensor networks for habitat monitoring. In *Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications*, pages 88–97, 2002.

[22] D. Malan, T. Fulford-jones, M. Welsh, and S. Moulton. Codeblue: An ad hoc sensor network infrastructure for emergency medical care. In *International Workshop on Wearable and Implantable Body Sensor Networks*, 2004.

[23] G. Mathur, P. Desnoyers, D. Ganesan, and P. Shenoy. Ultra-low power data storage for sensor networks. In *Proceedings of the 5th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 374–381, 2006.

[24] Microchip. 32-bit PIC MCUs. http://www.microchip.com/en_US/family/pic32/.

[25] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard. Quality of service profiling. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 25–34, 2010.

[26] V. Papirla and C. Chakrabarti. Energy-aware error control coding for flash memories. In *Proceedings of the 46th Annual Design Automation Conference (DAC)*, pages 658–663. ACM/EDAC/IEEE, 2009.

[27] P. Pavan, R. Bez, P. Olivo, and E. Zanoni. Flash memory cells-an overview. *Proceedings of the IEEE*, 85(8):1248–1271, Aug 1997.

[28] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling ultra-low power wireless research. In *Proceedings of the Fourth International Conference on Information Processing in Sensor Networks: Special track on Platform Tools and Design Methods for Network Embedded Sensors (IPSN/SPOTS)*, April 2005.

[29] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[30] R. L. Rivest. The RC5 encryption algorithm. In B. Preneel, editor, *Fast Software Encryption*, pages 86–96. Springer, 1995. (Proceedings Second International Workshop, Dec. 1994, Leuven, Belgium).

[31] R. L. Rivest and A. Shamir. How to reuse a write-once memory. *Information and Control*, 55:1–19, 1982.

[32] M. Salajegheh, S. Clark, B. Ransford, K. Fu, and A. Juels. CCCP: Secure remote storage for computational RFIDs. In *Proceedings of the 18th USENIX Security Symposium*, August 2009.

[33] A. P. Sample, D. J. Yeager, P. S. Powledge, A. V. Mamishev, and J. R. Smith. Design of an RFID-based battery-free programmable sensing platform. In *IEEE Transactions on Instrumentation and Measurement*, 2008.

[34] V. Shnayder, B.-r. Chen, K. Lorincz, T. R. F. F. Jones, and M. Welsh. Sensor networks for medical care. In *Proceedings of the 3rd ACM Conference on Embedded Networked Sensor Systems (SenSys)*, pages 314–314, 2005.

[35] I. Tamo and M. Schwartz. Correcting limited-magnitude errors in the rank-modulation scheme. *IEEE Transaction on Information Theory*, 56(6):2551–2560, 2010.

[36] Texas Instruments Incorporated. MSP430 Ultra-Low Power Microcontrollers. http://www.ti.com/msp430.

[37] C. Weddle, M. Oldham, J. Qian, A.-I. A. Wang, P. Reiher, and G. Kuenning. PARAID: A gear-shifting power-aware RAID. *ACM Transactions on Storage (TOS)*, 3(3):Article 13:1–33, 2007.

[38] D. Zeinalipour-Yazti, S. Lin, V. Kalogeraki, D. Gunopulos, and W. A. Najjar. Microhash: An efficient index structure for fash-based sensor devices. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies*, pages 31–44, 2005.

[39] G. Zemor and G. D. Cohen. Error-correcting WOM-codes. *IEEE Transactions on Information Theory*, 37(3):730–734, May 1991.

[40] F. Zhang, H. D. Pster, and A. Jiang. LDPC codes for rank modulation in flash memories. In *Proc. IEEE International Symposium on Information Theory (ISIT)*, 2010.

Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory

Shivaram Venkataraman$^{\dagger *}$*, Niraj Tolia*‡*, Parthasarathy Ranganathan*†*, and Roy H. Campbell**
†*HP Labs, Palo Alto,* ‡*Maginatics, and* **University of Illinois, Urbana-Champaign*

Abstract

The predicted shift to non-volatile, byte-addressable memory (e.g., Phase Change Memory and Memristor), the growth of "big data", and the subsequent emergence of frameworks such as memcached and NoSQL systems require us to rethink the design of data stores. To derive the maximum performance from these new memory technologies, this paper proposes the use of single-level data stores. For these systems, where no distinction is made between a volatile and a persistent copy of data, we present Consistent and Durable Data Structures (CDDSs) that, on current hardware, allows programmers to safely exploit the low-latency and non-volatile aspects of new memory technologies. CDDSs use versioning to allow atomic updates without requiring logging. The same versioning scheme also enables rollback for failure recovery. When compared to a memory-backed Berkeley DB B-Tree, our prototype-based results show that a CDDS B-Tree can increase put and get throughput by 74% and 138%. When compared to Cassandra, a two-level data store, Tembo, a CDDS B-Tree enabled distributed Key-Value system, increases throughput by up to 250%–286%.

1 Introduction

Recent architecture trends and our conversations with memory vendors show that DRAM density scaling is facing significant challenges and will hit a scalability wall beyond 40nm [26, 33, 34]. Additionally, power constraints will also limit the amount of DRAM installed in future systems [5, 19]. To support next generation systems, including large memory-backed data stores such as memcached [18] and RAMCloud [38], technologies such as Phase Change Memory [40] and Memristor [48] hold promise as DRAM replacements. Described in Section 2, these memories offer latencies that are comparable to DRAM and are orders of magnitude faster than either disk or flash. Not only are they byte-addressable and low-latency like DRAM but, they are also non-volatile.

Projected cost [19] and power-efficiency characteristics of Non-Volatile Byte-addressable Memory (NVBM) lead us to believe that it can replace both disk and memory in data stores (e.g., memcached, database systems, NoSQL systems, etc.) but not through legacy interfaces (e.g., block interfaces or file systems). First, the overhead of PCI accesses or system calls will dominate NVBM's sub-microsecond access latencies. More importantly, these interfaces impose a two-level logical separation of data, differentiating between in-memory and on-disk copies. Traditional data stores have to both update the in-memory data and, for durability, sync the data to disk with the help of a write-ahead log. Not only does this data movement use extra power [5] and reduce performance for low-latency NVBM, the logical separation also reduces the usable capacity of an NVBM system.

Instead, we propose a single-level NVBM hierarchy where no distinction is made between a volatile and a persistent copy of data. In particular, we propose the use of Consistent and Durable Data Structures (CDDSs) to store data, a design that allows for the creation of logless systems on non-volatile memory without processor modifications. Described in Section 3, these data structures allow mutations to be safely performed directly (using loads and stores) on the single copy of the data and metadata. We have architected CDDSs to use versioning. Independent of the update size, versioning allows the CDDS to atomically move from one consistent state to the next, without the extra writes required by logging or shadow paging. Failure recovery simply restores the data structure to the most recent consistent version. Further, while complex processor changes to support NVBM have been proposed [14], we show how primitives to provide durability and consistency can be created using existing processors.

We have implemented a CDDS B-Tree because of its non-trivial implementation complexity and widespread use in storage systems. Our evaluation, presented in Section 4, shows that a CDDS B-Tree can increase put and get throughput by 74% and 138% when compared to a memory-backed Berkeley DB B-Tree. Tembo[1], our Key-Value (KV) store described in Section 3.5, was created by integrating this CDDS B-Tree into a widely-used open-source KV system. Using the Yahoo Cloud Serving Benchmark [15], we observed that Tembo increases throughput by up to 250%–286% when compared to memory-backed Cassandra, a two-level data store.

2 Background and Related Work

2.1 Hardware Non-Volatile Memory

Significant changes are expected in the memory industry. Non-volatile flash memories have seen widespread adoption in consumer electronics and are starting to gain adoption in the enterprise market [20]. Recently, new NVBM memory technologies (e.g., PCM, Memristor, and STTRAM) have been demonstrated that significantly improve latency and energy efficiency compared to flash.

As an illustration, we discuss Phase Change Memory (PCM) [40], a promising NVBM technology. PCM is a non-volatile memory built out of Chalcogenide-based materials (e.g., alloys of germanium, antimony, or tellurium). Unlike DRAM and flash that record data through charge storage, PCM uses distinct phase change material states (corresponding to resistances) to store values. Specifically, when heated to a high temperature for an extended period of time, the materials crystallize and reduce their resistance. To reset the resistance, a current large enough to melt the phase change material is applied for a short period and then abruptly cut-off to quench the material into the amorphous phase. The two resistance states correspond to a '0' and '1', but, by varying the pulse width of the reset current, one can partially crystallize the phase change material and modify the resistance to an intermediate value between the '0' and '1' resistances. This range of resistances enables multiple bits per cell, and the projected availability of these MLC designs is 2012 [25].

Table 1 summarizes key attributes of potential storage alternatives in the next decade, with projected data from recent publications, technology trends, and direct industry communication. These trends suggest that future non-volatile memories such as PCM or Memristors can be viable DRAM replacements, achieving competitive speeds with much lower power consumption, and with non-volatility properties similar to disk but without the power overhead. Additionally, a number of recent studies have identified a slowing of DRAM growth [25, 26, 30, 33, 34, 39, 55] due to scaling challenges for charge-based memories. In conjunction with DRAM's power inefficiencies [5, 19], these trends can potentially accelerate the adoption of NVBM memories.

NVBM technologies have traditionally been limited by density and endurance, but recent trends suggest that these limitations can be addressed. Increased density can be achieved within a single-die through multi-level designs, and, potentially, multiple-layers per die. At a single chip level, 3D die stacking using through-silicon vias (TSVs) for inter-die communication can further increase density. PCM and Memristor also offer higher endurance than flash (10^8 writes/cell compared to 10^5 writes/cell for flash). Optimizations at the technology, circuit, and systems levels have been shown to further address endurance issues, and more improvements are likely as the technologies mature and gain widespread adoption.

These trends, combined with the attributes summarized in Table 1, suggest that technologies like PCM and Memristors can be used to provide a single "unified data-store" layer - an assumption underpinning the system architecture in our paper. Specifically, we assume a storage system layer that provides disk-like functionality but with memory-like performance characteristics and improved energy efficiency. This layer is persistent and byte-addressable. Additionally, to best take advantage of the low-latency features of these emerging technologies, non-volatile memory is assumed to be accessed off the memory bus. Like other systems [12, 14], we also assume that the hardware can perform atomic 8 byte writes.

While our assumed architecture is future-looking, it must be pointed out that many of these assumptions are being validated individually. For example, PCM samples are already available (e.g., from Numonyx) and an HP/Hynix collaboration [22] has been announced to bring Memristor to market. In addition, aggressive capacity roadmaps with multi-level cells and stacking have been discussed by major memory vendors. Finally, previously announced products have also allowed non-volatile memory, albeit flash, to be accessed through the memory bus [46].

2.2 File Systems

Traditional disk-based file systems are also faced with the problem of performing atomic updates to data structures. File systems like WAFL [23] and ZFS [49] use shadowing to perform atomic updates. Failure recovery in these systems is implemented by restoring the file system to a consistent snapshot that is taken periodically. These snapshots are created by shadowing, where every change to a block creates a new copy of the block. Recently, Rodeh [42] presented a B-Tree construction that can provide efficient support for shadowing and this tech-

[1] Swahili for elephant, an animal anecdotally known for its memory.

Technology	Density	Read/Write Latency		Read/Write Energy		Endurance
	um^2/bit	ns		pJ/bit		writes/bit
HDD	0.00006	3,000,000	3,000,000	2,500	2,500	∞
Flash SSD (SLC)	0.00210	25,000	200,000	250	250	10^5
DRAM (DIMM)	0.00380	55	55	24	24	10^{18}
PCM	0.00580	48	150	2	20	10^8
Memristor	0.00580	100	100	2	2	10^8

Table 1: Non-Volatile Memory Characteristics: 2015 Projections

nique has been used in the design of BTRFS [37]. Failure recovery in a CDDS uses a similar notion of restoring the data structure to the most recent consistent version. However the versioning scheme used in a CDDS results in fewer data-copies when compared to shadowing.

2.3 Non-Volatile Memory-based Systems

The use of non-volatile memory to improve performance is not new. eNVy [54] designed a non-volatile main memory storage system using flash. eNVy, however, accessed memory on a page-granularity basis and could not distinguish between temporary and permanent data. The Rio File Cache [11, 32] used battery-backed DRAM to emulate NVBM but it did not account for persistent data residing in volatile CPU caches. Recently there have been many efforts [21] to optimize data structures for flash memory based systems. FD-Tree [31] and BufferHash [2] are examples of write-optimized data structures designed to overcome high-latency of random writes, while FAWN [3] presents an energy efficient system design for clusters using flash memory. However, design choices that have been influenced by flash limitations (e.g., block addressing and high-latency random writes) render these systems suboptimal for NVBM.

Qureshi et al. [39] have also investigated combining PCM and DRAM into a hybrid main-memory system but do not use the non-volatile features of PCM. While our work assumes that NVBM wear-leveling happens at a lower layer [55], it is worth noting that versioning can help wear-leveling as frequently written locations are aged out and replaced by new versions. Most closely related is the work on NVTM [12] and BPFS [14]. NVTM, a more general system than CDDS, adds STM-based [44] durability to non-volatile memory. However, it requires adoption of an STM-based programming model. Further, because NVTM only uses a metadata log, it cannot guarantee failure atomicity. BPFS, a PCM-based file system, also proposes a single-level store. However, unlike CDDS's exclusive use of existing processor primitives, BPFS depends on extensive hardware modifications to provide correctness and durability. Further, unlike the data structure interface proposed in this work, BPFS implements a file system interface. While this is transparent to legacy applications, the system-call overheads reduce NVBM's low-latency benefits.

2.4 Data Store Trends

The growth of "big data" [1] and the corresponding need for scalable analytics has driven the creation of a number of different data stores today. Best exemplified by NoSQL systems [9], the throughput and latency requirements of large web services, social networks, and social media-based applications have been driving the design of next-generation data stores. In terms of storage, high-performance systems have started shifting from magnetic disks to flash over the last decade. Even more recently, this shift has accelerated to the use of large memory-backed data stores. Examples of the latter include memcached [18] clusters over 200 TB in size [28], memory-backed systems such as RAMCloud [38], in-memory databases [47, 52], and NoSQL systems such as Redis [41]. As DRAM is volatile, these systems provide data durability using backend databases (e.g., memcached/MySQL), on-disk logs (e.g., RAMCloud), or, for systems with relaxed durability semantics, via periodic checkpoints. We expect that these systems will easily transition from being DRAM-based with separate persistent storage to being NVBM-based.

3 Design and Implementation

As mentioned previously, we expect NVBM to be exposed across a memory bus and not via a legacy disk interface. Using the PCI interface (256 ns latency [24]) or even a kernel-based syscall API (89.2 and 76.4 ns for POSIX `read/write`) would add significant overhead to NVBM's access latencies (50–150 ns). Further, given the performance and energy cost of moving data, we believe that all data should reside in a single-level store where no distinction is made between volatile and persistent storage and all updates are performed in-place. We therefore propose that data access should use userspace libraries and APIs that map data into the process's address space.

However, the same properties that allow systems to take full advantage of NVBM's performance properties also introduce challenges. In particular, one of the biggest obstacles is that current processors do not provide primitives to order memory writes. Combined with the fact that the memory controller can reorder writes (at a cache line granularity), current mechanisms for updating data structures are likely to cause corruption in the face of power or software failures. For example, assume that a hash table insert requires the write of a new hash table object and is followed by a pointer write linking the new object to the hash table. A reordered write could propagate the pointer to main memory before the object and a failure at this stage would cause the pointer to link to an undefined memory region. Processor modifications for ordering can be complex [14], do not show up on vendor roadmaps, and will likely be preceded by NVBM availability.

To address these issues, our design and implementation focuses on three different layers. First, in Section 3.1, we describe how we implement ordering and flushing of data on existing processors. However, this low-level primitive is not sufficient for atomic updates larger than 8 bytes. In addition, we therefore also require versioning CDDSs, whose design principles are described in Section 3.2. After discussing our CDDS B-Tree implementation in Section 3.3 and some of the open opportunities and challenges with CDDS data structures in Section 3.4, Section 3.5 describes Tembo, the system resulting from the integration of our CDDS B-Tree into a distributed Key-Value system.

3.1 Flushing Data on Current Processors

As mentioned earlier, today's processors have no mechanism for preventing memory writes from reaching memory and doing so for arbitrarily large updates would be infeasible. Similarly, there is no guarantee that writes will not be reordered by either the processor or by the memory controller. While processors support a `mfence` instruction, it only provides write visibility and does not guarantee that all memory writes are propagated to memory (NVBM in this case) or that the ordering of writes is maintained. While cache contents can be flushed using the `wbinvd` instruction, it is a high-overhead operation (multiple ms per invocation) and flushes the instruction cache and other unrelated cached data. While it is possible to mark specific memory regions as write-through, this impacts write throughput as all stores have to wait for the data to reach main memory.

To address this problem, we use a combination of tracking recently written data and use of the `mfence` and `clflush` instructions. `clflush` is an instruction that invalidates the cache line containing a given memory address from all levels of the cache hierarchy, across multiple processors. If the cache line is dirty (i.e., it has uncommitted data), it is written to memory before invalidation. The `clflush` instruction is also ordered by the `mfence` instruction. Therefore, to commit a series of memory writes, we first execute an `mfence` as a barrier to them, execute a `clflush` on every cacheline of all modified memory regions that need to be committed to persistent memory, and then execute another `mfence`. In this paper, we refer to this instruction sequence as a *`flush`*. As microbenchmarks in Section 4.2 show, using `flush` will be acceptable for most workloads.

While this description and tracking dirty memory might seem complex, this was easy to implement in practice and can be abstracted away by macros or helper functions. In particular, for data structures, all updates occur behind an API and therefore the process of `flush`ing data to non-volatile memory is hidden from the programmer. Using the simplified hash table example described above, the implementation would first write the object and `flush` it. Only after this would it write the pointer value and then `flush` again. This two-step process is transparent to the user as it occurs inside the insert method.

Finally, one should note that while `flush` is necessary for durability and consistency, it is not sufficient by itself. If any metadata update (e.g., rebalancing a tree) requires an atomic update greater than the 8 byte atomic write provided by the hardware, a failure could leave it in an inconsistent state. We therefore need the versioning approach described below in Sections 3.2 and 3.3.

3.2 CDDS Overview

Given the challenges highlighted at the beginning of Section 3, an ideal data store for non-volatile memory must have the following properties:

- **Durable**: The data store should be durable. A fail-stop failure should not lose committed data.
- **Consistent**: The data store should remain consistent after every update operation. If a failure occurs during an update, the data store must be restored to a consistent state before further updates are applied.
- **Scalable**: The data store should scale to arbitrarily-large sizes. When compared to traditional data stores, any space, performance, or complexity overhead should be minimal.
- **Easy-to-Program**: Using the data store should not introduce undue complexity for programmers or unreasonable limitations to its use.

We believe it is possible to meet the above properties by storing data in Consistent and Durable Data Structures (CDDSs), i.e., hardened versions of conventional data structures currently used with volatile memory. The ideas used in constructing a CDDS are applicable to a wide variety of linked data structures and, in this paper, we implement a CDDS B-Tree because of its non-trivial implementation complexity and widespread use in storage systems. We would like to note that the design and implementation of a CDDS only addresses *physical* consistency, i.e., ensuring that the data structure is readable

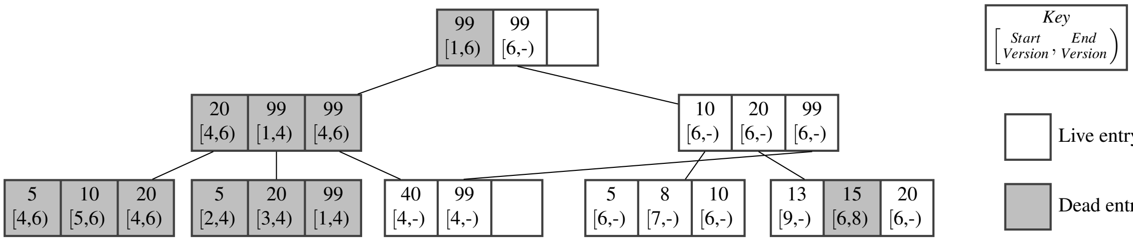


Figure 1: Example of a CDDS B-Tree

and never left in a corrupt state. Higher-level layers control *logical* consistency, i.e., ensuring that the data stored in the data structure is valid and matches external integrity constraints. Similarly, while our current system implements a simple concurrency control scheme, we do not mandate concurrency control to provide isolation as it might be more efficient to do it at a higher layer.

A CDDS is built by maintaining a limited number of versions of the data structure with the constraint that an update should not weaken the structural integrity of an older version and that updates are atomic. This versioning scheme allows a CDDS to provide consistency without the additional overhead of logging or shadowing. A CDDS thus provides a guarantee that a failure between operations will never leave the data in an inconsistent state. As a CDDS never acknowledges completion of an update without safely committing it to non-volatile memory, it also ensures that there is no silent data loss.

3.2.1 Versioning for Durability

Internally, a CDDS maintains the following properties:

- There exists a version number for the most recent consistent version. This is used by any thread which wishes to read from the data structure.
- Every update to the data structure results in the creation of a new version.
- During the update operation, modifications ensure that existing data representing older versions are never overwritten. Such modifications are performed by either using atomic operations or copy-on-write style changes.
- After all the modifications for an update have been made persistent, the most recent consistent version number is updated atomically.

3.2.2 Garbage Collection

Along with support for multiple versions, a CDDS also tracks versions of the data structure that are being accessed. Knowing the oldest version which has a non-zero reference count has two benefits. First, we can garbage collect older versions of the data structure. Garbage collection (GC) is run in the background and helps limit the space utilization by eliminating data that will not be referenced in the future. Second, knowing the oldest active version can also improve performance by enabling intelligent space reuse in a CDDS. When creating a new entry, the CDDS can proactively reclaim the space used by older inactive versions.

3.2.3 Failure Recovery

Insert or delete operations may be interrupted due to operating system crashes or power failures. By definition, the most recent consistent version of the data structure should be accessible on recovery. However, an in-progress update needs to be removed as it belongs to an uncommitted version. We handle failures in a CDDS by using a 'forward garbage collection' procedure during recovery. This process involves discarding all update operations which were executed after the most recent consistent version. New entries created can be discarded while older entries with in-progress update operations are reverted.

3.3 CDDS B-Trees

As an example of a CDDS, we selected the B-Tree [13] data structure because of its widespread use in databases, file systems, and storage systems. This section discusses the design and implementation of a consistent and durable version of a B-Tree. Our B-Tree modifications[2] have been heavily inspired by previous work on multi-version data structures [4, 50]. However, our focus on durability required changes to the design and impacted our implementation. We also do not retain all previous versions of the data structure and can therefore optimize updates.

In a CDDS B-Tree node, shown in Figure 1, the key and value stored in a B-Tree entry is augmented with a start and end version number, represented by unsigned 64-bit integers. A B-Tree node is considered 'live' if it has at least one live entry. In turn, an entry is considered 'live' if it does not have an end version (displayed as a '−' in the figure). To bound space utilization, in addition to ensuring that a minimum number of entries in a B-Tree node are used, we also bound the minimum number of

[2] In reality, our B-Tree is a B+ Tree with values only stored in leaves.

Algorithm 1: CDDS B-Tree Lookup

```
   Input: k: key, r: root
   Output: val: value
1  begin lookup (k, r)
2      $v \leftarrow current\_version$
3      $n \leftarrow r$
4      while is_inner_node ($n$) do
5          $entry\_num \leftarrow$ find ($k, n, v$)
6          $n \leftarrow n[entry\_num].child$
7      $entry\_num \leftarrow$ find ($k, n, v$)
8      return $n[entry\_num].value$
9  end
10 begin find ($k, n, v$)
11     $l \leftarrow 0$
12     $h \leftarrow$ get_num_entries ($n$)
13     while $l < h$ do                          // Binary Serch
14         $m \leftarrow (l+h)/2$
15         if $k \leq n[m].key$ then
16             $h \leftarrow m-1$
17         else $l \leftarrow m+1$
18     while $h <$ get_num_entries ($n$) do
19         if $n[h].start \leq v$ then
20             if $n[h].end > v$ || $n[h].end = 0$ then
21                 break
22         $h \leftarrow h+1$
23     return $h$
24 end
```

live entries in each node. Thus, while the CDDS B-Tree API is identical to normal B-Trees, the implementation differs significantly. In the rest of this section, we use the *lookup*, *insert*, and *delete* operations to illustrate how the CDDS B-Tree design guarantees consistency and durability[3].

3.3.1 Lookup

We first briefly describe the lookup algorithm, shown in Algorithm 1. For ease of explanation and brevity, the pseudocode in this and following algorithms does not include all of the design details. The algorithm uses the `find` function to recurse down the tree (lines 4–6) until it finds the leaf node with the correct key and value.

Consider a lookup for the key 10 in the CDDS B-Tree shown in Figure 1. After determining the most current version (version 9, line 2), we start from the root node and pick the rightmost entry with key 99 as it is the next largest valid key. Similarly in the next level, we follow the link from the leftmost entry and finally retrieve the value for 10 from the leaf node.

Our implementation currently optimizes lookup performance by ordering node entries by key first and then by the start version number. This involves extra writes during inserts to shift entries but improves read performance by enabling a binary search within nodes (lines 13–17 in `find`). While we have an alternate implementation that optimizes writes by not ordering keys at the cost of higher lookup latencies, we do not use it as our target workloads are read-intensive. Finally, once we detect the right index in the node, we ensure that we are returning a version that was valid for v, the requested version number (lines 18–22).

Algorithm 2: CDDS B-Tree Insertion

```
   Input: k: key, r: root
1  begin insert_key ($k, r$)
2      $v \leftarrow current\_version$
3      $v' \leftarrow v+1$
       // Recurse to leaf node (n)
4      $y \leftarrow$ get_num_entries ($n$)
5      if $y = node\_size$ then                    // Node Full
6          if $entry\_num$ = can_reuse_version ($n$,$y$) then
7              $n[entry\_num].key \leftarrow k$
8              $n[entry\_num].start \leftarrow v'$
9              $n[entry\_num].end \leftarrow 0$
10             flush ($n[entry\_num]$)
11         else
12             split_insert ($n, k, v'$)
               // Update inner nodes
13     else
14         $n[y].key \leftarrow k$
15         $n[y].start \leftarrow v'$
16         $n[y].end \leftarrow 0$
17         flush ($n[y]$)
18     $current\_version \leftarrow v'$
19     flush ($current\_version$)
20 end
21 begin split_insert ($n, k, v$)
22     $l \leftarrow$ num_live_entries ($n$)
23     $m_l \leftarrow$ min_live_entries
24     if $l > 4 * m_l$ then
25         $nn_1 \leftarrow$ new_node
26         $nn_2 \leftarrow$ new_node
27         for $i = 1$ to $l/2$ do
28             insert ($nn_1, n[i].key, v$)
29         for $i = l/2+1$ to $l$ do
30             insert ($nn_2, n[i].key, v$)
31         if $k < n[l/2].key$ then
32             insert ($nn_1, k, v$)
33         else insert ($nn_2, k, v$)
34         flush ($nn_1, nn_2$)
35     else
36         $nn \leftarrow$ new_node
37         for $i = 1$ to $l$ do
38             insert ($nn, n[i].key, v$)
39         insert ($nn, k, v$)
40         flush ($nn$)
41     for $i = 1$ to $l$ do
42         $n[i].end \leftarrow v$
43     flush ($n$)
44 end
```

[3] A longer technical report [51] presents more details on all CDDS B-Tree operations and their corresponding implementations.

3.3.2 Insertion

The algorithm for inserting a key into a CDDS B-Tree is shown in Algorithm 2. Our implementation of the algorithm uses the `flush` operation (described in Section 3.1) to perform atomic operations on a cacheline. Consider the case where a key, 12, is inserted into the B-Tree shown in Figure 1. First, an algorithm similar to lookup is used to find the leaf node that contains the key range that 12 belongs to. In this case, the right-most leaf node is selected. As shown in lines 2–3, the current consistent version is read and a new version number is generated. As the leaf node is full, we first use the `can_reuse_version` function to check if an existing dead entry can be reused. In this case, the entry with key 15 died at version 8 and is reused. To reuse a slot we first remove the key from the node and shift the entries to maintain them in sorted order. Now we insert the new key and again shift entries as required. For each key shift, we ensure that the data is first `flush`ed to another slot before it is overwritten. This ensures that the safety properties specified in Section 3.2.1 are not violated. While not described in the algorithm, if an empty entry was detected in the node, it would be used and the order of the keys, as specified in Section 3.3.1, would be maintained.

If no free or dead entry was found, a `split_insert`, similar to a traditional B-Tree split, would be performed. `split_insert` is a copy-on-write style operation in which existing entries are copied before making a modification. As an example, consider the node shown in Figure 2, where the key 40 is being inserted. We only need to preserve the 'live' entries for further updates and `split_insert` creates one or two new nodes based on the number of live entries present. Note that setting the end version (lines 41–42) is the only change made to the existing leaf node. This ensures that older data versions are not affected by failures. In this case, two new nodes are created at the end of the split.

The inner nodes are now updated with links to the newly created leaf nodes and the parent entries of the now-dead nodes are also marked as dead. A similar procedure is followed for inserting entries into the inner nodes. When the root node of a tree overflows, we split the node using the `split_insert` function and create one or two new nodes. We then create a new root node with links to the old root and to the newly created split-nodes. The pointer to the root node is updated atomically to ensure safety.

Once all the changes have been `flush`ed to persistent storage, the current consistent version is update atomically (lines 18–19). At this point, the update has been successfully committed to the NVBM and failures will not result in the update being lost.

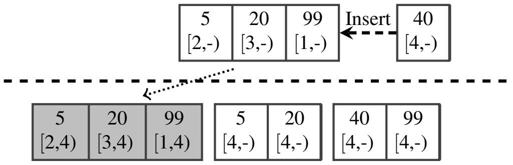


Figure 2: CDDS node split during insertion

Algorithm 3: CDDS B-Tree Deletion

```
   Input: k: key, r: root
1  begin delete (k, r)
2      $v \leftarrow current\_version$
3      $v' \leftarrow v+1$
       // Recurse to leaf node (n)
4      $y \leftarrow$ find_entry (n, k)
5      $n[y].end \leftarrow v'$
6      $l \leftarrow$ num_live_entries (n)
7      if $l = m_l$ then                              // Underflow
8          $s \leftarrow$ pick_sibling (n)
9          $l_s \leftarrow$ num_live_entries (s)
10         if $l_s > 3 \times m_l$ then
11             copy_from_sibling (n, s, v′)
12         else merge_with_sibling (n, s, v′)
           // Update inner nodes
13     else flush (n[y])
14     $current\_version \leftarrow v'$
15     flush (current_version)
16 end
17 begin merge_with_sibling (n, s, v)
18     $y \leftarrow$ get_num_entries (s)
19     if $y < 4 \times m_l$ then
20         for $i = 1$ to $m_l$ do
21             insert $(s, n[i].key, v)$
22             $n[i].end \leftarrow v$
23     else
24         $nn \leftarrow$ new_node
25         $l_s \leftarrow$ num_live_entries (s)
26         for $i = 1$ to $l_s$ do
27             insert $(nn, s[i].key, v)$
28             $s[i].end \leftarrow v$
29         for $i = 1$ to $m_l$ do
30             insert $(nn, n[i].key, v)$
31             $n[i].end \leftarrow v$
32         flush (nn)
33     flush $(n, s)$
34 end
35 begin copy_from_sibling (n, s, v)
       // Omitted for brevity
36 end
```

3.3.3 Deletion

Deleting an entry is conceptually simple as it simply involves setting the end version number for the given key. It does not require deleting any data as that is handled by GC. However, in order to bound the number of live blocks in the B-Tree and improve space utilization, we shift live entries if the number of live entries per node reaches m_l, a threshold defined in Section 3.3.6. The only exception is the root node as, due to a lack of siblings, shifting within the same level is not feasible. However,

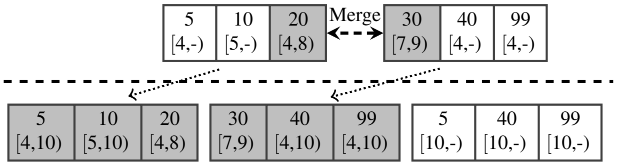


Figure 3: CDDS node merge during deletion

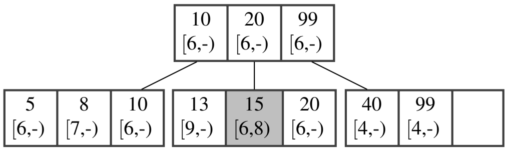


Figure 4: CDDS B-Tree after Garbage Collection

as described in Section 3.3.4, if the root only contains one live entry, the child will be promoted.

As shown in Algorithm 3, we first check if the sibling has at least $3 \times m_l$ live entries and, if so, we copy m_l live entries from the sibling to form a new node. As the leaf has m_l live entries, the new node will have $2 \times m_l$ live entries. If that is not the case, we check if the sibling has enough space to copy the live entries. Otherwise, as shown in Figure 3, we merge the two nodes to create a new node containing the live entries from the leaf and sibling nodes. The number of live entries in the new node will be $\geq 2 \times m_l$. The inner nodes are updated with pointers to the newly created nodes and, after the changes have been flushed to persistent memory, the current consistent version is incremented.

3.3.4 Garbage Collection

As shown in Section 3.3.3, the size of the B-Tree does not decrease when keys are deleted and can increase due to the creation of new nodes. To reduce the space overhead, we therefore use a periodic GC procedure, currently implemented using a mark-and-sweep garbage collector [8]. The GC procedure first selects the latest version number that can be safely garbage collected. It then starts from the root of the B-Tree and deletes nodes which contain dead and unreferenced entries by invalidating the parent pointer to the deleted node. If the root node contains only one live entry after garbage collection, the child pointed to by the entry is promoted. This helps reduce the height of the B-Tree. As seen in the transformation of Figure 1 to the reduced-height tree shown in Figure 4, only live nodes are present after GC.

3.3.5 Failure Recovery

The recovery procedure for the B-Tree is similar to garbage collection. In this case, nodes newer than the more recent consistent version are removed and older nodes are recursively analyzed for partial updates. The recovery function performs a physical 'undo' of these updates and ensures that the tree is physically and logically identical to the most recent consistent version. While our current recovery implementation scans the entire data structure, the recovery process is fast as it operates at memory bandwidth speeds and only needs to verify CDDS metadata.

3.3.6 Space Analysis

In the CDDS B-Tree, space utilization can be characterized by the number of live blocks required to store N key-value pairs. Since the values are only stored in the leaf nodes, we analyze the maximum number of live leaf nodes present in the tree. In the CDDS B-Tree, a new node is created by an insert or delete operation. As described in Sections 3.3.2 and 3.3.3, the minimum number of live entries in new nodes is $2 \times m_l$.

When the number of live entries in a node reaches m_l, it is either merged with a sibling node or its live entries are copied to a new node. Hence, the number of live entries in a node is $> m_l$. Therefore, in a B-Tree with N live keys, the maximum number of live leaf nodes is bound by $O(\frac{N}{m_l})$. Choosing m_l as $\frac{k}{5}$, where k is the size of a B-Tree node, the maximum number of live leaf nodes is $O(\frac{5N}{k})$.

For each live leaf node, there is a corresponding entry in the parent node. Since the number of live entries in an inner node is also $> m_l$, the number of parent nodes required is $O\left(\frac{\frac{5N}{k}}{m_l}\right) = O(\frac{N}{(\frac{k}{5})^2})$. Extending this, we can see that the height of the CDDS B-Tree is bound by $O(log_{\frac{k}{5}}N)$. This also bounds the time for all B-Tree operations.

3.4 CDDS Discussion

Apart from the CDDS B-Tree operations described above, the implementation also supports additional features including iterators and range scans. We believe that CDDS versioning also lends itself to other powerful features such as instant snapshots, rollback for programmer recovery, and integrated NVBM wear-leveling. We hope to explore these issues in our future work.

We also do not anticipate the design of a CDDS preventing the implementation of different concurrency schemes. Our current CDDS B-Tree implementation uses a multiple-reader, single-writer model. However, the use of versioning lends itself to more complex concurrency control schemes including multi-version concurrency control (MVCC) [6]. While beyond the scope of this paper, exploring different concurrency control schemes for CDDSs is a part of our future work.

CDDS-based systems currently depend on virtual memory mechanisms to provide fault-isolation and like other services, it depends on the OS for safety. Therefore, while unlikely, placing NVBM on the memory bus can expose it to accidental writes from rogue DMAs. In contrast, the narrow traditional block device interface makes it harder to accidentally corrupt data. We believe that hardware memory protection, similar to IOMMUs, will be required to address this problem. Given that we map data into an application's address space, stray writes from a buggy application could also destroy data. While this is no different from current applications that `mmap` their data, we are developing lightweight persistent heaps that use virtual memory protection with a RVM-style API [43] to provide improved data safety.

Finally, apart from multi-version data structures [4, 50], CDDSs have also been influenced by Persistent Data Structures (PDSs) [17]. The "Persistent" in PDS does not actually denote durability on persistent storage but, instead, represents immutable data structures where an update always yields a new data structure copy and never modifies previous versions. The CDDS B-Tree presented above is a weakened form of semi-persistent data structures. We modify previous versions of the data structure for efficiency but are guaranteed to recover from failure and rollback to a consistent state. However, the PDS concepts are applicable, in theory, to all linked data structures. Using PDS-style techniques, we have implemented a proof-of-concept CDDS hash table and, as evidenced by previous work for functional programming languages [35], we are confident that CDDS versioning techniques can be extended to a wide range of data structures.

3.5 Tembo: A CDDS Key-Value Store

We created Tembo, a CDDS Key-Value (KV) store, to evaluate the effectiveness of a CDDS-based data store. The system involves the integration of the CDDS-based B-Tree described in Section 3.3 into Redis [41], a widely used event-driven KV store. As our contribution is not based around the design of this KV system, we only briefly describe Tembo in this section. As shown in Section 4.4, the integration effort was minor and leads us to believe that retrofitting CDDS into existing applications will be straightforward.

The base architecture of Redis is well suited for a CDDS as it retains the entire data set in RAM. This also allows an unmodified Redis to serve as an appropriate performance baseline. While persistence in the original system was provided by a write-ahead append-only log, this is eliminated in Tembo because of the CDDS B-Tree integration. For fault-tolerance, Tembo provides master-slave replication with support for hierarchical replication trees where a slave can act as the master for other replicas. Consistent hashing [27] is used by client libraries to distribute data in a Tembo cluster.

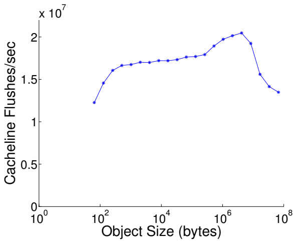


Figure 5: Flushes/second

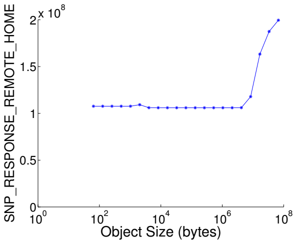


Figure 6: Cache Snooping

4 Evaluation

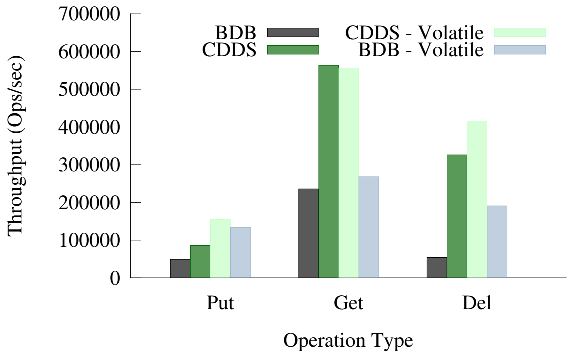
In this section, we evaluate our design choices in building Consistent and Durable Data Structures. First, we measure the overhead associated with techniques used to achieve durability on existing processors. We then compare the CDDS B-tree to Berkeley DB and against log-based schemes. After briefly discussing CDDS implementation and integration complexity, we present results from a multi-node distributed experiment where we use the Yahoo Cloud Serving Benchmark (YCSB) [15].

4.1 Evaluation Setup

As NVBM is not commercially available yet, we used DRAM-based servers. While others [14] have shown that DRAM-based results are a good predictor of NVBM performance, as a part of our ongoing work, we aim to run micro-architectural simulations to confirm this within the context of our work. Our testbed consisted of 15 servers with two Intel Xeon Quad-Core 2.67 GHz (X5550) processors and 48 GB RAM each. The machines were connected via a full-bisection Gigabit Ethernet network. Each processor has 128 KB L1, 256 KB L2, and 8 MB L3 caches. While each server contained 8 300 GB 10K SAS drives, unless specified, all experiments were run directly on RAM or on a ramdisk. We used the Ubuntu 10.04 Linux distribution and the 2.6.32-24 64-bit kernel.

4.2 Flush Performance

To accurately capture the performance of the `flush` operation defined in Section 3.1, we used the "MultCallFlushLRU" methodology [53]. The experiment allocates 64 MB of memory and subdivides it into equally-sized cache-aligned objects. Object sizes ranged from 64 bytes to 64 MB. We write to every cache line in an object, `flush` the entire object, and then repeat the process with the next object. For improved timing accuracy, we stride over the memory region multiple times.



Mean of 5 trials. Max. standard deviation: 2.2% of the mean.

Figure 7: Berkeley DB Comparison

Remembering that each `flush` is a number of `clflush`es bracketed by `mfence`s on both sides, Figure 5 shows the number of `clflush`es executed per second. Flushing small objects sees the worst performance (∼12M cacheline flushes/sec for 64 byte objects). For larger objects (256 bytes–8 MB), the performance ranges from ∼16M–20M cacheline flushes/sec.

We also observed an unexpected drop in performance for large objects (>8 MB). Our analysis showed that this was due to the cache coherency protocol. Large objects are likely to be evicted from the L3 cache before they are explicitly flushed. A subsequent `clflush` would miss in the local cache and cause a high-latency "snoop" request that checks the second off-socket processor for the given cache line. As measured by the UNC_SNP_RESP_TO_REMOTE_HOME.I_STATE performance counter, seen in Figure 6, the second socket shows a corresponding spike in requests for cache lines that it does not contain. To verify this, we physically removed a processor and observed that the anomaly disappeared[4]. Further, as we could not replicate this slowdown on AMD platforms, we believe that cache-coherency protocol modifications can address this anomaly.

Overall, the results show that we can `flush` 0.72–1.19 GB/s on current processors. For applications without networking, Section 4.3 shows that future hardware support can help but applications using `flush` can still outperform applications that use file system `sync` calls. Distributed applications are more likely to encounter network bottlenecks before `flush` becomes an overhead.

4.3 API Microbenchmarks

This section compares the CDDS B-Tree performance for puts, gets, and deletes to Berkeley DB's (BDB) B-Tree implementation [36]. For this experiment, we insert, fetch, and then delete 1 million key-value tuples into each system. After each operation, we flush the CPU cache to eliminate any variance due to cache contents. Keys and values are 25 and 2048 bytes large. The single-threaded benchmark driver runs in the same address space as BDB and CDDS. BDB's cache size was set to 8 GB and could hold the entire data set in memory. Further, we configure BDB to maintain its log files on an in-memory partition.

	Lines of Code
Original STX B-Tree	2,110
CDDS Modifications	1,902
Redis (v2.0.0-rc4)	18,539
Tembo Modifications	321

Table 2: Lines of Code Modified

We run both CDDS and BDB (v4.8) in durable and volatile modes. For BDB volatile mode, we turn transactions and logging off. For CDDS volatile mode, we turn `flush`ing off. Both systems in volatile mode can lose or corrupt data and would not be used where durability is required. We only present the volatile results to highlight predicted performance if hardware support was available and to discuss CDDS design tradeoffs.

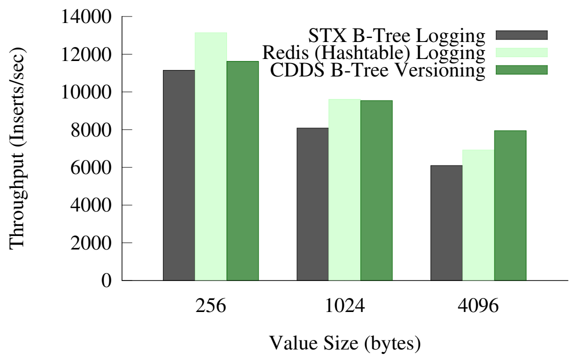
The results, displayed in Figure 7, show that, for memory-backed BDB in durable mode, the CDDS B-Tree improves throughout by 74%, 138%, and 503% for puts, gets, and deletes respectively. These gains come from not using a log (extra writes) or the file system interface (system call overhead). CDDS delete improvement is larger than puts and gets because we do not delete data immediately but simply mark it as dead and use GC to free unreferenced memory. In results not presented here, reducing the value size, and therefore the log size, improves BDB performance but CDDS always performs better.

If zero-overhead epoch-based hardware support [14] was available, the CDDS volatile numbers show that performance of puts and deletes would increase by 80% and 27% as `flush`es would never be on the critical path. We do not observe any significant change for gets as the only difference between the volatile and durable CDDS is that the `flush` operations are converted into a noop.

We also notice that while volatile BDB throughput is lower than durable CDDS for gets and dels by 52% and 41%, it is higher by 56% for puts. Puts are slower for the CDDS B-Tree because of the work required to maintain key ordering (described in Section 3.3.1), GC overhead, and a slightly higher height due to nodes with a mixture of live and dead entries. Volatile BDB throughput is also higher than durable BDB but lower than volatile CDDS for all operations.

Finally, to measure versioning overhead, we compared the volatile CDDS B-Tree to a normal B-Tree [7]. While not presented in Figure 7, volatile CDDS's performance

[4] We did not have physical access to the experimental testbed and ran the processor removal experiment on a different dual-socket Intel Xeon (X5570) machine.



Mean of 5 trials. Max. standard deviation: 6.7% of the mean.

Figure 8: Versioning vs. Logging

was lower than the in-memory B-Tree by 24%, 13%, and 39% for puts, gets, and dels. This difference is similar to other performance-optimized versioned B-trees [45].

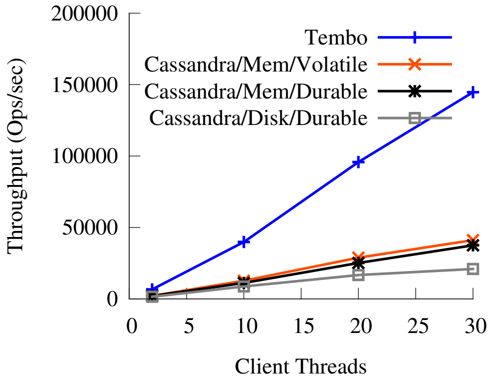
4.4 Implementation Effort

The CDDS B-Tree started with the STX C++ B-Tree [7] implementation but, as measured by `sloccount` and shown in Table 2, the addition of versioning and NVBM durability replaced 90% of the code. While the API remained the same, the internal implementation differs substantially. The integration with Redis to create Tembo was simpler and only changed 1.7% of code and took less than a day to integrate. Since the CDDS B-Tree implements an interface similar to an STL Sorted Container, we believe that integration with other systems should also be simple. Overall, our experiences show that while the initial implementation complexity is moderately high, this only needs to be done once for a given data structure. The subsequent integration into legacy or new systems is straightforward.

4.5 Tembo Versioning vs. Redis Logging

Apart from the B-Tree specific logging performed by BDB in Section 4.3, we also wanted to compare CDDS versioning when integrated into Tembo to the write-ahead log used by Redis in fully-durable mode. Redis uses a hashtable and, as it is hard to compare hashtables and tree-based data structures, we also replaced the hashtable with the STX B-Tree. In this single-node experiment, we used 6 Tembo or Redis data stores and 2 clients[5]. The write-ahead log for the Redis server was stored on an in-memory partition mounted as `tmpfs` and did not use the hard disk. Each client performed 1M inserts over the loopback interface.

The results, presented in Figure 8, show that as the value size is increased, Tembo performs up to 30% better than Redis integrated with the STX B-Tree. While Redis updates the in-memory data copy and also writes to the append-only log, Tembo only updates a single copy. While hashtable-based Redis is faster than Tembo for 256 byte values because of faster lookups, even with the disadvantage of a tree-based structure, Tembo's performance is almost equivalent for 1 KB values and is 15% faster for 4 KB values.



Mean of 5 trials. Max. standard deviation: 7.8% of the mean.

Figure 9: YCSB: SessionStore

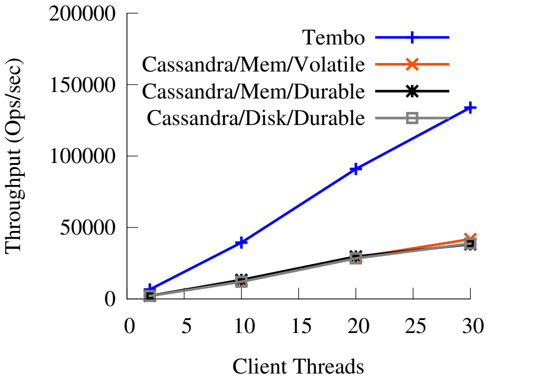
The results presented in this section are lower than the improvements in Section 4.3 because of network latency overhead. The `fsync` implementation in `tmpfs` also does not explicitly flush modified cache lines to memory and is therefore biased against Tembo. We are working on modifications to the file system that will enable a fairer comparison. Finally, some of the overhead is due to maintaining ordering properties in the CDDS-based B-Tree to support range scans - a feature not used in the current implementation of Tembo.

4.6 End-to-End Comparison

For an end-to-end test, we used YCSB, a framework for evaluating the performance of Key-Value, NoSQL, and cloud storage systems [15]. In this experiment, we used 13 servers for the cluster and 2 servers as the clients. We extended YCSB to support Tembo, and present results from two of YCSB's workloads. Workload-A, referred to as SessionStore in this section, contains a 50:50 read:update mix and is representative of tracking recent actions in an online user's session. Workload-D, referred to as StatusUpdates, has a 95:5 read:insert mix. It represents people updating their online status (e.g., Twitter tweets or Facebook wall updates) and other users reading them. Both workloads execute 2M operations on values consisting of 10 columns with 100 byte fields.

We compare Tembo to Cassandra (v0.6.1) [29], a distributed data store that borrows concepts from BigTable [10] and Dynamo [16]. We used three different Cassandra configurations in this experiment. The

[5] Being event-driven, both Redis and Tembo are single-threaded. Therefore one data store (or client) is run per core in this experiment.



Mean of 5 trials. Max. standard deviation: 8.1% of the mean.

Figure 10: YCSB: StatusUpdates

first two used a ramdisk for storage but the first (Cassandra/Mem/Durable) flushed its commit log before every update while the second (Cassandra/Mem/Volatile) only flushed the log every 10 seconds. For completeness, we also configured Cassandra to use a disk as the backing store (Cassandra/Disk/Durable).

Figure 9 presents the aggregate throughput for the SessionStore benchmark. With 30 client threads, Tembo's throughput was 286% higher than memory-backed durable Cassandra. Given Tembo and Cassandra's different design and implementation choices, the experiment shows the overheads of Cassandra's in-memory "memtables," on-disk "SSTables," and a write-ahead log, vs. Tembo's single-level store. Disk-backed Cassandra's throughput was only 22–44% lower than the memory-backed durable configuration. The large number of disks in our experimental setup and a 512 MB battery-backed disk controller cache were responsible for this better-than-expected disk performance. On a different machine with fewer disks and a smaller controller cache, disk-backed Cassandra bottlenecked with 10 client threads.

Figure 10 shows that, for the StatusUpdates workload, Tembo's throughput is up to 250% higher than memory-backed durable Cassandra. Tembo's improvement is slightly lower than the SessionStore benchmark because StatusUpdates insert operations update all 10 columns for each value, while the SessionStore only selects one random column to update. Finally, as the entire data set can be cached in memory and inserts represent only 5% of this workload, the different Cassandra configurations have similar performance.

5 Conclusion and Future Work

Given the impending shift to non-volatile byte-addressable memory, this work has presented Consistent and Durable Data Structures (CDDSs), an architecture that, without processor modifications, allows for the creation of log-less storage systems on NVBM. Our results show that redesigning systems to support single-level data stores will be critical in meeting the high-throughput requirements of emerging applications.

We are currently also working on extending this work in a number of directions. First, we plan on leveraging the inbuilt CDDS versioning to support multi-version concurrency control. We also aim to explore the use of relaxed consistency to further optimize performance as well as integration with virtual memory to provide better safety against stray application writes. Finally, we are investigating the integration of CDDS versioning and wear-leveling for better performance.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd Michael A. Kozuch for their valuable feedback. We would also like to thank Jichuan Chang, Antonio Lain, Jeff Mogul, Craig Soules, and Robert E. Tarjan for their feedback on earlier drafts of this paper. We would also like to thank Krishnan Narayan and Eric Wu for their help with our experimental infrastructure.

References

[1] The data deluge. *The Economist*, 394(8671):11, Feb. 2010.

[2] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath. Cheap and large cams for high performance data-intensive networked systems. In *Proceedings of the 7th Symposium on Networked Systems Design and Implementation (NSDI '10)*, pages 433–448, San Jose, CA, Apr. 2010.

[3] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP 2009)*, pages 1–14, Big Sky, MT, Oct. 2009.

[4] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion b-tree. *The VLDB Journal*, 5(4):264–275, 1996.

[5] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, K. Yelick, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally,

M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Keckler, D. Klein, P. Kogge, R. S. Williams, and K. Yelick. Exascale computing study: Technology challenges in achieving exascale systems, 2008. DARPA IPTO, ExaScale Computing Study, http://users.ece.gatech.edu/mrichard/ExascaleComputingStudyReports/ECS_reports.htm.

[6] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, 1981.

[7] T. Bingmann. STX B+ Tree, Sept. 2008. http://idlebox.net/2007/stx-btree/.

[8] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software: Practices and Experience*, 18(9):807–820, 1988.

[9] R. Cattell. High performance data stores. http://www.cattell.net/datastores/Datastores.pdf, Apr. 2010.

[10] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[11] P. M. Chen, W. T. Ng, S. Chandra, C. M. Aycock, G. Rajamani, and D. E. Lowell. The rio file cache: Surviving operating system crashes. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 74–83, Cambridge, MA, Oct. 1996.

[12] J. Coburn, A. Caulfield, L. Grupp, A. Akel, and S. Swanson. NVTM: A transactional interface for next-generation non-volatile memories. Technical Report CS2009-0948, University of California, San Diego, Sept. 2009.

[13] D. Comer. Ubiquitous b-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.

[14] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. C. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 133–146, Big Sky, MT, Oct. 2009.

[15] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*, pages 143–154, Indianapolis, IN, June 2010.

[16] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*, pages 205–220, Stevenson, WA, 2007.

[17] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.

[18] B. Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004(124):5, 2004.

[19] R. F. Freitas and W. W. Wilcke. Storage-class memory: The next storage system technology. *IBM Journal of Research and Development*, 52(4):439–447, 2008.

[20] FusionIO, Sept. 2010. http://www.fusionio.com/.

[21] E. Gal and S. Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys*, 37:138–163, June 2005.

[22] Hewlett-Packard Development Company. HP Collaborates with Hynix to Bring the Memristor to Market in Next-generation Memory, Aug. 2010. http://www.hp.com/hpinfo/newsroom/press/2010/100831c.html.

[23] D. Hitz, J. Lau, and M. Malcolm. File system design for an nfs file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 19–19, San Francisco, California, 1994.

[24] B. Holden. Latency comparison between hypertransport and pci-express in communications systems. Whitepaper, Nov. 2006.

[25] International Technology Roadmap for Semiconductors, 2009. http://www.itrs.net/Links/2009ITRS/Home2009.htm.

[26] International Technology Roadmap for Semiconductors: Process integration, Devices, and Structures, 2007. http://www.itrs.net/Links/2007ITRS/2007_Chapters/2007_PIDS.pdf.

[27] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the 29th Annual ACM Symposium on*

Theory of Computing (STOC '97), pages 654–663, El Paso, TX, 1997.

[28] M. Kwiatkowski. memcache@facebook, Apr. 2010. QCon Beijing 2010 Enterprise Software Development Conference. `http://www.qconbeijing.com/download/marc-facebook.pdf`.

[29] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[30] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*, pages 2–13, Austin, TX, 2009.

[31] Y. Li, B. He, Q. Luo, and K. Yi. Tree indexing on flash disks. In *Proceedings of the 25th IEEE International Conference on Data Engineering (ICDE)*, pages 1303–1306, Washington, DC, USA, Apr. 2009.

[32] D. E. Lowell and P. M. Chen. Free transactions with rio vista. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 92–101, St. Malo, France, Oct. 1997.

[33] J. A. Mandelman, R. H. Dennard, G. B. Bronner, J. K. DeBrosse, R. Divakaruni, Y. Li, and C. J. Radens. Challenges and future directions for the scaling of dynamic random-access memory (DRAM). *IBM Journal of Research and Development*, 46(2-3):187–212, 2002.

[34] W. Mueller, G. Aichmayr, W. Bergner, E. Erben, T. Hecht, C. Kapteyn, A. Kersch, S. Kudelka, F. Lau, J. Luetzen, A. Orth, J. Nuetzel, T. Schloesser, A. Scholz, U. Schroeder, A. Sieck, A. Spitzer, M. Strasser, P.-F. Wang, S. Wege, and R. Weis. Challenges for the DRAM cell scaling to 40nm. In *IEEE International Electron Devices Meeting*, pages 339–342, May 2005.

[35] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, July 1999. ISBN 0521663504.

[36] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference*, pages 183–191, Monterey, CA, June 1999.

[37] Oracle Corporation. BTRFS, June 2009. `http://btrfs.wiki.kernel.org`.

[38] J. K. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for RAMClouds: Scalable high-performance storage entirely in DRAM. *ACM SIGOPS Operating Systems Review*, 43:92–105, January 2010.

[39] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA 2009)*, pages 24–33, Austin, TX, June 2009.

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

[41] Redis, Sept. 2010. `http://code.google.com/p/redis/`.

[42] O. Rodeh. B-trees, shadowing, and clones. *ACM Transactions on Storage (TOS)*, 3:2:1–2:27, February 2008.

[43] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. *ACM Transactions on Computer Systems*, 12(1):33–57, 1994.

[44] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 204–213, Ottawa, Canada, Aug. 1995.

[45] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger. Metadata efficiency in versioning file systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03)*, pages 43–58, San Francisco, CA, Mar. 2003.

[46] Spansion, Inc. Using spansion ecoram to improve tco and power consumption in internet data centers, 2008. `http://www.spansion.com/jp/About/Documents/Spansion_EcoRAM_Architecture_J.pdf`.

[47] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (its time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*, pages 1150–1160, Vienna, Austria, Sept. 2007.

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

[49] Sun Microsystems. ZFS, Nov. 2005. http://www.opensolaris.org/os/community/zfs/.

[50] P. J. Varman and R. M. Verma. An efficient multiversion access structure. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):391–409, 1997.

[51] S. Venkataraman and N. Tolia. Consistent and durable data structures for non-volatile byte-addressable memory. Technical Report HPL-2010-110, HP Labs, Palo Alto, CA, Sept. 2010.

[52] VoltDB, Sept. 2010. http://www.voltdb.com/.

[53] R. C. Whaley and A. M. Castaldo. Achieving accurate and context-sensitive timing for code optimization. *Software – Practice and Experience*, 38(15): 1621–1642, 2008.

[54] M. Wu and W. Zwaenepoel. eNVy: A non-volatile, main memory storage system. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 86–97, San Jose, CA, Oct. 1994.

[55] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, pages 14–23, Austin, TX, June 2009.

CAFTL: A Content-Aware Flash Translation Layer Enhancing the Lifespan of Flash Memory based Solid State Drives

Feng Chen* Tian Luo Xiaodong Zhang

Dept. of Computer Science & Engineering
The Ohio State University
Columbus, OH 43210, USA
{*fchen,luot,zhang*}@*cse.ohio-state.edu*

Abstract

Although Flash Memory based Solid State Drive (SSD) exhibits high performance and low power consumption, a critical concern is its limited lifespan along with the associated reliability issues. In this paper, we propose to build a Content-Aware Flash Translation Layer (CAFTL) to enhance the endurance of SSDs at the device level. With no need of any semantic information from the host, CAFTL can effectively reduce write traffic to flash memory by removing unnecessary duplicate writes and can also substantially extend available free flash memory space by coalescing redundant data in SSDs, which further improves the efficiency of garbage collection and wear-leveling. In order to retain high data access performance, we have also designed a set of acceleration techniques to reduce the runtime overhead and minimize the performance impact caused by extra computational cost. Our experimental results show that our solution can effectively identify up to 86.2% of the duplicate writes, which translates to a write traffic reduction of up to 24.2% and extends the flash space by a factor of up to 31.2%. Meanwhile, CAFTL only incurs a minimized performance overhead by a factor of up to 0.5%.

1 Introduction

The limited lifespan is the *Achilles' heel* of Flash Memory based Solid State Drives (SSDs). On one hand, SSDs built on semiconductor chips without any moving parts have exhibited many unique technical merits compared with hard disk drives (HDDs), particularly high random access performance and low power consumption. On the other hand, the limited lifespan of SSDs, which are built on flash memories with limited erase/program cycles, is still one of the most critical concerns that seriously hinder a wide deployment of SSDs in reliability-sensitive environments, such as data centers [10]. Although SSD manufacturers often claim that SSDs can sustain routine usage for years, the technical concerns about the endurance issues of SSDs still remain high. This is mainly due to three not-so-well-known reasons. First, as bit density increases, flash memory chips become more affordable but, at the same time, less reliable and less durable. In the last two years, for high-density flash devices, we have seen a sharp drop of erase/program cycle ratings from ten thousand to five thousand cycles [7]. As technology scaling continues, this situation could become even worse. Second, traditional redundancy solutions such as RAID, which have been widely used for battling disk failures, are considered less effective for SSDs, because of the high probability of correlated device failures in SSD-based RAID [9]. Finally, although some prior research work [13, 22, 33] has presented empirical and modeling-based studies on the lifespan of flash memories and USB flash drives, both positive and negative results have been reported. In fact, as a recent report from Google® points out, "endurance and retention (of SSDs) not yet proven in the field" [10].

All these aforesaid issues explain why commercial users hesitate to perform a large-scale deployment of SSDs in production systems and why integrating SSDs into commercial systems is proceeding such "painfully slowly" [10]. In order to integrate such a "frustrating technology", which comes with equally outstanding merits and limits, into the existing storage hierarchy timely and reliably, solutions for effectively improving the lifespan of SSDs are highly desirable. In this paper, we propose such a solution from a unique and viable angle.

1.1 Background of SSDs

1.1.1 Flash memory and SSD internals

NAND flash memory is the basic building block of most SSDs on the market. A flash memory package is usually composed of one or multiple *dies* (chips). Each die is segmented into multiple *planes*, and a plane is further divided into thousands (e.g. 2048) of *erase blocks*. An erase block usually consists of 64-128 *pages*. Each page has a data area (e.g. 4KB) and a spare area (a.k.a. metadata area). Flash memories support three major operations. *Read* and *write* (a.k.a. *program*) are performed in units of pages, and *erase*, which clears all the pages in an erase block, must be conducted in erase blocks.

*Currently working at the Intel Labs in Hillsboro, OR.

Flash memory has three critical technical constraints: (1) *No in-place overwrite* – the whole erase block must be erased before writing (programming) any page in this block. (2) *No random writes* – the pages in an erase block must be written sequentially. (3) *Limited erase/program cycles* – an erase block can wear out after a certain number of erase/program cycles (typically 10,000-100,000).

As a critical component in the SSD design, the *Flash Translation Layer* (FTL) is implemented in the SSD controller to emulate a hard disk drive by exposing an array of *logical block addresses* (LBAs) to the host. In order to address the aforesaid three constraints, the FTL designers have developed several sophisticated techniques: (1) *Indirect mapping* – A mapping table is maintained to track the dynamic mapping between logical block addresses (LBAs) and physical block addresses (PBAs). (2) *Log-like write mechanism* – Each write to a logical page only invalidates the previously occupied physical page, and the new content data is appended sequentially in a clean erase block, like a *log*, which is similar to the log-structured file system [41]. (3) *Garbage collection* – A garbage collector (GC) is launched periodically to recycle invalidated physical pages, consolidate the valid pages into a new erase block, and clean the old erase block. (4) *Wear-leveling* – Since writes are often concentrated on a subset of data, which may cause some blocks to wear out earlier than the others, a *wear-leveling* mechanism tracks and shuffles hot/cold data to even out writes in flash memory. (5) *Over-provisioning* – In order to assist garbage collection and wear-leveling, SSD manufacturers usually include a certain amount of over-provisioned spare flash memory space in addition to the host-usable SSD capacity.

1.1.2 The lifespan of SSDs

As flash memory has a limited number of erase/program cycles, the lifespan of SSDs is naturally constrained. In essence, the lifespan of SSDs is a function of three factors: (1) *The amount of incoming write traffic* – The less data written into an SSD, the longer the lifespan would be. In fact, the SSD manufacturers often advise commercial users, whose systems undergo intensive write traffic (e.g. an email server), to purchase more expensive high-end SSDs. (2) *The size of over-provisioned flash space* – A larger over-provisioned flash space provides more available clean flash pages in the allocation pool that can be used without triggering a garbage collection. Aggressive over-provisioning can effectively reduce the average number of writes over all flash pages, which in turn improves the endurance of SSDs. For example, the high-end Intel® X25-E SSD is aggressively over-provisioned with about 8GB flash space, which is 25% of the labeled SSD capacity (32GB) [25]. (3) *The efficiency of garbage collection and wear-leveling mechanisms* – Having been extensively researched, the garbage collection and wear-leveling policies can significantly impact the lifespan of SSDs. For example, *static wear-leveling*, which swaps active blocks with randomly chosen inactive blocks, performs better in endurance than *dynamic wear-leveling*, which only swaps active blocks [13].

Most previous research work [21] focuses on the third factor, garbage collection and wear-leveling policies. A survey [21] summarizes these techniques. In contrast, little study has been conducted on the other two aspects. This may be because incoming write traffic is normally believed to be workload dependent, which cannot be changed at the device level, and the over-provisioning of flash space is designated at the manufacturing process and cannot be excessively large (due to the production cost). In this paper we will show that even at the SSD device level, we can still effectively extend the SSD lifespan by reducing the amount of incoming write traffic and squeezing available flash memory space during runtime, which has not been considered before. This goal can be achieved based on our observation of a widely existing phenomenon – *data duplication*.

1.2 Data Duplication is Common

In file systems data duplication is very common. For example, kernel developers can have multiple versions of Linux source code for different projects. Users can create/delete the same files multiple times. Another example is word editing tools, which often automatically save a copy of documents every few minutes, and the content of these copies can be almost identical.

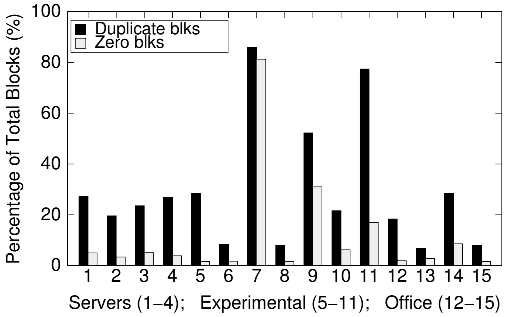


Figure 1: The percentage of redundant data in disks.

To make a case here, we have studied 15 disks installed on 5 machines in the Department of Computer Science and Engineering at the Ohio State University. Three file systems can be found in these disks, namely Ext2, Ext3, and NTFS. The disks are used in different environments, 4 disks from *Database/Web Servers*, 7 disks from *Experimental Systems* for kernel development, and the other 4 disks from *Office Systems*. We slice the disk space into 4KB blocks and use the SHA-1 hash function [1] to calculate a 160-bit hash value for each block. We can identify duplicate blocks by comparing the hash values. Figure 1 shows the *duplication rates* (i.e. the percentage of duplicate blocks in total blocks).

In Figure 1, we find that the duplication rate ranges from 7.9% to 85.9% across the 15 disks. We also find that in only one disk with NTFS, the duplicate blocks are dominated by 'zero' blocks. The duplicate blocks on the other disks are mostly non-zero blocks, which means that these duplicate blocks contain 'meaningful' data. Considering the fact that a typical SSD has an over-provisioned space of only 1-20% of the flash memory space, removing the duplicate data, which accounts for 7.9-85.9% of the SSD capacity, can substantially extend the available flash space that can be used for garbage collection and wear-leveling. If this effort is successful, we can raise the performance comparable to that of high-end SSDs with no need of extra flash space.

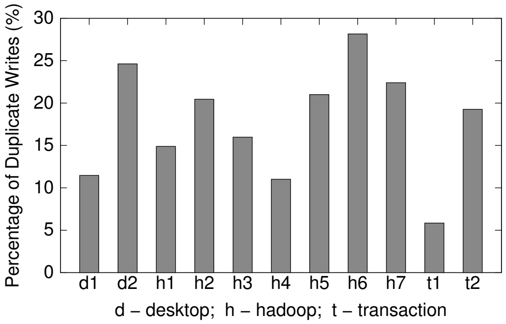


Figure 2: The perc. of duplicate writes in workloads.

Besides the static analysis of the data redundancy in storage, we have also collected I/O traces and analyzed the data accesses of 11 workloads from three categories (see more details in Section 4). For each workload, we modified the Linux kernel by intercepting each I/O request and calculating a hash value for each requested block. We analyzed the I/O traces off-line. Figure 2 shows the percentage of the duplicate writes in each workload. We can find that 5.8-28.1% of the writes are duplicated. This finding suggests that if we remove these duplicate writes, we can effectively reduce the write traffic into flash medium, which directly improves the endurance accordingly, not to mention the indirect effect of reducing the number of extra writes caused by less frequently triggered garbage collections.

1.3 Making FTL Content Aware

Based on the above observations and analysis, we propose a *Content-Aware Flash Translation Layer* (CAFTL) to integrate the functionality of eliminating duplicate writes and redundant data into SSDs to enhance the lifespan at the device level.

CAFTL intercepts incoming write requests at the SSD device level and uses a collision-free cryptographic hash function to generate fingerprints summarizing the content of updated data. By querying a fingerprint store, which maintains the fingerprints of resident data in the SSD, CAFTL can accurately and safely eliminate duplicate writes to flash medium. CAFTL also uses a two-level mapping mechanism to coalesce redundant data, which effectively extends available flash space and improves GC efficiency. In order to minimize the performance impact caused by computing hash values, we have also designed a set of acceleration methods to speed up fingerprinting. With these techniques, CAFTL can effectively reduce write traffic to flash, extend available flash space, while retaining high data access performance.

CAFTL is an augmentation, rather than a complete replacement, to the existing FTL designs. Being *content-aware*, CAFTL is orthogonal to the other FTL policies, such as the well researched garbage collection and wear-leveling policies. In fact, the existing mechanisms in the SSDs provide much needed facilities for CAFTL and make it a perfect fit in the existing SSD architecture. For example, the *indirect mapping mechanism* naturally makes associating multiple logical pages to one physical page easy to implement; the *periodic scanning process* for garbage collection and wear-leveling can also carry out an out-of-line deduplication asynchronously; the *log-like write mechanism* makes it possible to re-validate the 'deleted' data without re-writing the same content; and finally, the *semiconductor nature* of flash memory makes reading randomly remapped data free of high latencies.

CAFTL is also backward compatible and portable. Running at the device level as a part of SSD firmware, CAFTL does not need to change the standard host/device interface for passing any extra information from the upper-level components (e.g. file system) to the device. All of the design of CAFTL is isolated at the device level and hidden from users. This guarantees CAFTL as a drop-in solution, which is highly desirable in practice.

1.4 Our Contributions

We have made the following contributions in this paper: (1) We have studied data duplications in file systems and various workloads, and assessed the viability of improving endurance of SSDs through deduplication. (2) We have carefully designed a content-aware FTL to extend the SSD lifespan by removing duplicate writes (up to 24.2%) and redundant data (up to 31.2%) with minimal overhead. To the best of our knowledge, this is the first study using effective deduplication in SSDs. (3) We have also designed a set of techniques to accelerate the in-line deduplication in SSD devices, which are particularly effective with small on-device buffer spaces (e.g. 2MB) and make performance overhead nearly negligible. (4) We have implemented CAFTL in the DiskSim simulator and comprehensively evaluated its performance and shown the effectiveness of improving the SSD lifespan through extensive trace-driven simulations.

The rest of this paper is organized as follows. In Section 2, we discuss the unique challenges in the design of CAFTL. Section 3 introduces the design of CAFTL and our acceleration methods. We present our performance evaluation in Section 4. Section 5 gives the related work. The last section discusses and concludes this paper.

2 Technical Challenges

CAFTL shares the same principle of removing data redundancy with Content-Addressable Storage (CAS), e.g. [11, 24, 30, 45, 47], which is designed for backup/archival systems. However, we cannot simply borrow CAS policies in our design due to four unique and unaddressed challenges: (1) *Limited resources* – CAFTL is designed for running in an SSD device with limited memory space and computing power, rather than running on a dedicated powerful enterprise server. (2) *Relatively lower redundancy* – CAFTL mostly handles regular file system workloads, which have an impressive but much lower duplication rate than that of backup streams with high redundancy (often 10 times or even higher). (3) *Lack of semantic hints* – CAFTL works at the device level and only sees a sequence of logical blocks without any semantic hints from host file systems. (4) *Low overhead requirement* – CAFTL must retain high data access performance for regular workloads, while this is a less stringent requirement in backup systems that can run during out-of-office hours.

All of these unique requirements make deduplication particularly challenging in SSDs and it requires nontrivial efforts to address them in the CAFTL design.

3 The Design of CAFTL

The design of CAFTL aims to reach the following three critical objectives.

- *Reducing unnecessary write traffic* – By examining the data of incoming write requests, we can detect and remove duplicate writes in-line, so that we can effectively filter unnecessary writes into flash memory and directly improve the lifespan of SSDs.
- *Extending available flash space* – By leveraging the indirect mapping framework in SSDs, we can map logical pages sharing the same content to the same physical page. The saved space can be used for GC and wear-leveling, which indirectly improves the lifespan.
- *Retaining access performance* – A critical requirement to make CAFTL truly effective in practice is to avoid significant negative performance impacts. We must minimize runtime overhead and retain high data access performance.

3.1 Overview of CAFTL

CAFTL eliminates duplicate writes and redundant data through a combination of both *in-line* and *out-of-line* (a.k.a post-processing or out-of-band) deduplication. Inline deduplication refers to the case where CAFTL proactively examines the incoming data and cancels duplicate writes before committing a write request to flash. As a ‘best-effort’ solution, CAFTL does not guarantee that all duplicate writes can be examined and removed immediately (e.g. it can be disabled for performance purposes). Thus CAFTL also periodically scans the flash memory and coalesces redundant data out of line.

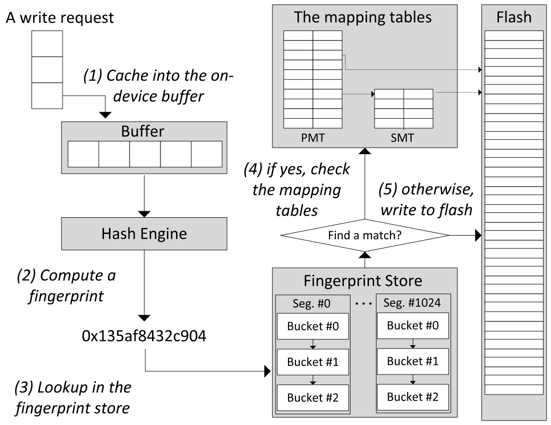


Figure 3: An illustration of CAFTL architecture.

Figure 3 illustrates the process of handling a write request in CAFTL – When a write request is received at the SSD, (1) the incoming data is first temporarily maintained in the *on-device buffer*; (2) each updated page in the buffer is later computed a hash value, also called *fingerprint*, by a *hash engine*, which can be a dedicated processor or simply a part of the controller logic; (3) each fingerprint is looked up against a *fingerprint store*, which maintains the fingerprints of data already stored in the flash memory; (4) if a match is found, which means that a residing data unit holds the same content, the *mapping tables*, which translate the host-viewable logical addresses to the physical flash addresses, are updated by mapping it to the physical location of the residing data, and correspondingly the write to flash is canceled; (5) if no match is found, the write is performed to the flash memory as a regular write.

3.2 Hashing and Fingerprint Store

CAFTL attempts to identify and remove duplicate writes and redundant data. A byte-by-byte comparison is excessively slow. A common practice is to use a cryptographic hash function, e.g. SHA-1 [1] or MD5 [40], to compute a collision-free hash value as a fingerprint. Duplicate data can be determined by comparing fingerprints. Here we explain how we produce and manage fingerprints.

3.2.1 Choosing hashing units

CAFTL uses a chunk-based deduplication approach. Unlike most CAS systems, which often use more complicated *variable-sized* chunking, CAFTL adopts a *fixed-sized* chunking approach for two reasons. First, the variable-sized chunking is designed for segmenting a long I/O stream. In CAFTL, we handle a sequence of individual requests, whose size can be very small (a few kilobytes) and vary significantly. Thus variable-sized chunking is inappropriate for CAFTL. Second, the basic operation unit in flash is a page (e.g. 4KB), and the internal management policies in SSDs, such as the mapping policy, are also designed in units of pages. Thus, using pages as the fixed-sized chunks for hashing is a natural choice and also avoids unnecessary complexity.

3.2.2 Hash function and fingerprints

In order to identify duplicate data, a collision-free hash function is used for summarizing the content of pages. We use the SHA-1 [1], a widely used cryptographic hash function, and rely on its collision-resistant properties to index and compare pages. For each page, we calculate a 160-bit hash value as its *fingerprint* and store it as the page's metadata in flash. The SHA-1 hash function has been proven computationally infeasible to find two distinct inputs hashing to the same value [32]. We can safely determine if two pages are identical using fingerprints.

3.2.3 The fingerprint store

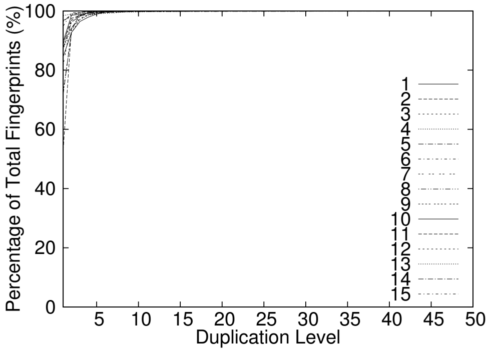


Figure 4: The CDF figure of duplicate fingerprints.

In order to locate quickly the physical page with a specific fingerprint, CAFTL manages an in-memory structure, called *Fingerprint Store*. Apparently, keeping all fingerprints and related information (25 bytes each) in memory is too costly and unnecessary. We have studied the distribution of fingerprints in the 15 disks and we plot a Cumulative Distribution Function (CDF) figure in Figure 4. We can see that the distribution of duplicated fingerprints is skewed – only 10-20% of the fingerprints are highly duplicated (more than 2). This finding provides two implications. First, most fingerprints are unique and never have a chance to match any queried fingerprint. Second, a complete search in the fingerprint store would incur high lookup latencies, and even worse, most lookups eventually turn out to be useless (no match found). Thus, we should only store and search in the most likely-to-be-duplicated fingerprints in memory.

We first logically partition the hash value space into N segments. For a given fingerprint, f, we can map it to segment $(f \ mod \ N)$, and the random nature of the hash function guarantees an even distribution of fingerprints among the segments. Each segment contains a list of *buckets*. Each bucket is a 4KB page in memory and consists of multiple entries, each of which is a key-value pair, $\{$*fingerprint*, (*location*, *reference*)$\}$. The 160-bit *fingerprint* indexes the entry; the 32-bit *location* denotes where we can find the data, either the PBA of a physical flash page or the VBA of a secondary mapping entry (see Section 3.3); the 8-bit *reference* denotes the hotness of this fingerprint (i.e. the number of referencing logical pages). The entries in each bucket are sorted in the ascending order of their fingerprint values to facilitate a fast in-bucket binary search. The total numbers of buckets and segments are designated by the SSD manufacturers.

The fingerprint store maintains the most highly referenced fingerprints in memory. During the SSD startup time, after the mapping tables are built up (to be discussed in Section 3.3), the fingerprint store is also reconstructed by scanning the mapping tables and the metadata in flash to load the key value pairs of $\{$*fingerprint*, (*location*, *reference*)$\}$ into memory. Initially no bucket is allocated in the fingerprint store. Upon inserting a fingerprint, an empty bucket is allocated and linked into a bucket list of the corresponding segment. This bucket holds the fingerprints inserted into the corresponding segment until the bucket is filled up, then we allocate another bucket. We continue to allocate buckets in this way until there are no more free buckets available. If that happens, the newly inserted fingerprint will replace the fingerprint with the smallest reference counter (i.e. the coldest one) in the bucket, unless its reference counter is smaller than any of the resident fingerprints. Note that we choose the inserting bucket in a round-robin manner to ensure a relatively even distribution of hot/cold fingerprints across the buckets in a segment. It is also worth mentioning here that a 8-bit reference counter is sufficiently large for distinguishing the hot fingerprints, because most fingerprints have a reference counter smaller than 255 (see Figure 4). We consider fingerprints with a reference counter larger than 255 as highly referenced and do not further distinguish their difference in hotness. In this way, we can include the most highly referenced fingerprints in memory. Although we may miss some opportunities of identifying the duplicates whose fingerprints are not resident in memory, this probability is considered low (as shown in Figure 4), and we are not pursuing a perfect in-line deduplication. Our out-of-line scanning can still identify these duplicates later.

Searching a fingerprint can be very simple. We compute the mapping segment number and scan the corresponding list of buckets one by one. In each bucket, we use binary search to speed up the in-bucket lookup. However, for a segment with a large set of buckets, this method is still improvable. We have designed three optimization techniques to further accelerate fingerprint lookups. (1) *Range Check* – before performing the binary search in a bucket, we first compare the fingerprint with the smallest and the largest fingerprints in the buckets. If the fingerprint is out of the range, we quickly skip over this bucket. (2) *Hotness-based Reorganization* – the fingerprints in the linked buckets can be reorganized in the descending order of their reference counters. This moves the hot fingerprints closer to the list head and potentially reduces the number of the scanned buckets. (3) *Bucket-level Binary Search* – the fingerprints across the buckets can be reorganized in the ascending order of the fingerprint values by using a merge sort. For each segment we maintain an array of pointers to the buckets in the list. We can perform a binary search at the bucket level by recursively selecting the bucket in the middle to do a Range Check. In this way we can quickly locate the target bucket and skip over most buckets. Although reorganizing the fingerprints requires performing an additional merge sort, our experiments show that these optimizations can significantly reduce the number of comparisons of fingerprint values. In Section 4.3.3 we will show and compare the effectiveness of the three techniques.

3.3 Indirect Mapping

Indirect mapping is a core mechanism in the SSD architecture. SSDs expose an array of logical block addresses (LBAs) to the host, and internally, a *mapping table* is maintained to track the physical block address (PBA) to which each LBA is mapped. For CAFTL, the existing indirect mapping mechanism in SSDs provides a basic framework for deduplication and avoids rebuilding the whole infrastructure from scratch.

On the other hand, the existing 1-to-1 mapping mechanism in SSDs cannot be directly used for CAFTL, which is essentially *N*-to-1 mapping, because of two new challenges. (1) When a physical page is relocated to another place (e.g. in garbage collection), we must be able to identify quickly all the logical pages mapped to this physical page and update their mapping entries to point to the new location. (2) Since a physical page could be shared by multiple logical pages, it cannot be recycled by the garbage collector until all the referencing logical pages are demapped from it, which means that we must track the number of referencing logical pages.

3.3.1 Two-level indirect mapping

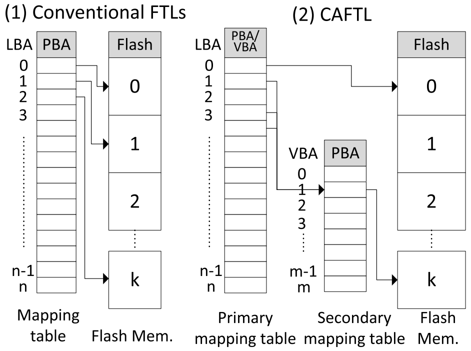


Figure 5: An illustration of the indirect mapping.

We have designed a new indirect mapping mechanism to address these aforementioned issues. As shown in Figure 5, a conventional FTL uses a one-level indirect mapping, from LBAs to PBAs. In CAFTL, we create another indirect mapping level, called *Virtual Block Addresses* (VBAs). A VBA is essentially a pseudo address name to represent a set of LBAs mapped to the same PBA. In this two-level indirect mapping structure, we can locate the physical page for a logical page either through LBA→PBA or LBA→VBA→PBA.

We maintain a *primary mapping table* and a *secondary mapping table* in memory. The *primary mapping table* maps a LBA to either a PBA, if the logical page is unique, or a VBA, if it is a duplicate page. We differentiate PBAs and VBAs by using the most significant bit in the 32-bit page address. For a page size of 4KB, using the remaining 31 bits can address 8,192 GB storage space, which is sufficiently large for an SSD. The *secondary mapping table* maps a VBA to a PBA. Each entry is indexed by the *VBA* and has two fields, $\{PBA, reference\}$. The 32-bit *PBA* denotes the physical flash page, and the 32-bit *reference* tracks the exact number of logical pages mapped to the physical page. Only physical pages without any reference can be recycled for garbage collection.

This two-level indirect mapping mechanism has several merits. First, it significantly simplifies the reverse updates to the mapping of duplicate logical pages. When relocating a physical page during GC, we can use its associated VBA to quickly locate and update the secondary mapping table by mapping the VBA to the new location (PBA), which avoids exhaustively searching for all the referencing LBAs in the huge primary mapping table. Second, the secondary mapping table can be very small. Since CAFTL handles regular file system workloads, most logical pages are unique and directly mapped through the primary table. We can also apply an approach similar to DFTL [23] to further reduce the memory demand by selectively maintaining the most frequently accessed entries of the mapping tables in memory. Finally, this incurs minimal additional lookup overhead. For unique pages, it performs identically to conventional FTLs; for duplicate pages, only one extra memory access is needed for the lookup operation.

3.3.2 The mapping tables in flash

The mapping relationship is also maintained in flash memory. We keep an in-flash copy of the primary and secondary mapping tables along with a *journal* in dedicated flash space in SSD. Both in-flash structures are organized as a list of linked physical flash pages. When updating the in-memory tables (e.g. remapping a LBA to a new location), the update record is logged into a small in-memory buffer. When the buffer is filled, the log records are appended to the in-flash journal. If power failure happens, a capacitor (e.g. a SuperCap [46]) can provide sufficient current to flush the unwritten logs into the journal and secure the critical mapping structures in persistent storage. Periodically the in-memory tables are synced into flash and the journal is reinitialized. During the startup time, the in-flash tables are first loaded into memory and the logged updates in the journal are applied to reconstruct the mapping tables.

3.3.3 The metadata pages in flash

Unlike much prior work, which writes the metadata (e.g. LBA and fingerprint) in the spare area of physical flash pages, we reserve a dedicated number of flash pages, also called *metadata pages*, to store the metadata, and keep a *metadata page array* for tracking PBAs of the metadata pages. The spare area of a physical page is only used for storing the Error Correction Code (ECC) checksum. If each physical page is associated with 24 bytes of metadata (a 160-bit fingerprint and a 32-bit LBA/VBA), for a 32GB SSD with 4KB flash pages, we need about 0.6% of the flash space for storing metadata and a 192KB metadata page array. In this way, we can detach the data pages and the metadata pages, which allows us to manage flexibly the metadata for physical flash pages.

3.4 Acceleration Methods

Fingerprinting is the key bottleneck of the in-line deduplication in CAFTL, especially when the on-device buffer size is limited. Here we present three effective techniques to reduce its negative performance impact.

3.4.1 Sampling for hashing

In file system workloads, as we discussed previously, duplicate writes are not a ‘common case’ as in backup systems. This means that most time we spend on fingerprinting is not useful at all. Thus, we selectively pick only one page as a *sample page* for fingerprinting, and we use this sample fingerprint to query the fingerprint store to see if we can find a match there. If this is true, the whole write request is very likely to be a duplicate, and we can further compute fingerprints for the other pages to confirm that. Otherwise, we assume the whole request would not be a duplicate and abort fingerprinting at the earliest time. In this way, we can significantly reduce the hashing cost.

The key issue here is which page should be chosen as the sample page. It is particularly challenging in CAFTL, since CAFTL only sees a sequence of blocks and cannot leverage any file-level semantic hints (e.g. [11]). We propose to use *Content-based Sampling* – We select the first four bytes, called *sample bytes*, from each page in a request, and we concatenate the four bytes into a 32-bit numeric value. We compare these values and the page with the *largest* value is the sample page. The rationale behind this is that if two requests carry similar content, the pages with the largest sample bytes in two requests would be very likely to be the same, too. We deliberately avoid selecting the sample pages based on hash values (e.g. [11,30]), because in CAFTL, hashing itself incurs high latency. Thus relying on hash values for sampling is undesirable, so we directly pick sample pages based on their *unprocessed* content data. We have also examined choosing other bytes (Figure 6) as the sample bytes and found that using the first four bytes performs constantly well across different workloads.

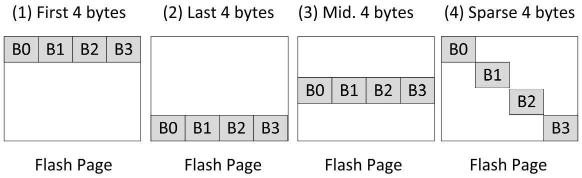


Figure 6: An illustration of four choices of sample bytes.

In our implementation of sampling, we divide the sequence of pages in a write request into several *sampling units* (e.g. 32 pages), and we pick one sample page from each unit. We also note that sampling could affect deduplication – the larger a sampling unit is, the better performance but the lower deduplication rate would be. We will study the effect of unit sizes in Section 4.4.1.

3.4.2 Light-weight pre-hashing

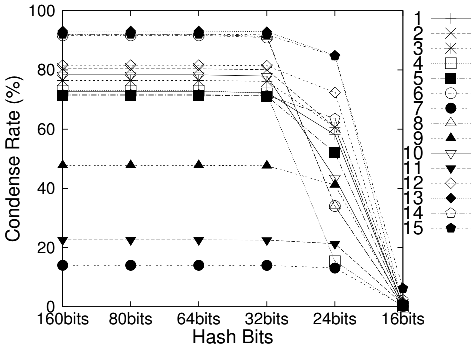


Figure 7: Condense rates vs. hash bits.

Computing a light-weight hash function often incurs lower computational cost. For example, producing a 32-bit CRC32 hash value is over 10 times faster than computing a 160-bit SHA-1 hash value. More importantly, our study shows that reducing the hash strength would *not* incur a significant increase of false positives for a typical SSD capacity. We can see in Figure 7 that using only 32 bits can achieve nearly the same condense rate as using 160 bits. Plus, many SSDs integrate a dedicated ECC engine to compute checksum and detect errors, which can also be leveraged to speed up hashing.

We propose a technique, called *light-weight prehashing*. We maintain an extra 32-bit CRC32 hash value for each fingerprint in the fingerprint store. For a page, we first compute a CRC32 hash value and query the fingerprint store. If a match is found, which means the page is very likely to be a duplicate, then we use the SHA-1 hash function to generate a fingerprint and confirm it in the fingerprint store; otherwise, we abort the high-cost SHA-1 fingerprinting immediately and perform the write to flash. Although maintaining CRC32 hash values demands more fingerprint store space, the significant performance benefit well justifies it, as shown in Section 4.4.2. We have also considered using a Bloom filter [12] for pre-screening, like in the DataDomain® file system [47], but found it inapplicable to CAFTL, because it requires multiple hashings and the summary vector cannot be updated when a fingerprint is removed.

3.4.3 Dynamic switches

In some extreme cases, incoming requests may wait for available buffer space to be released by previous requests. CAFTL provides *dynamic switch* as the last line of defense for performance protection in such cases.

We set a *high watermark* and a *low watermark* to turn the in-line deduplication off and on, respectively. If the percentage of the occupied cache space hits a high watermark (95%), we disable the in-line deduplication to flush writes quickly to flash and release buffer space. Once the low watermark (50%) is hit, we re-enable the in-line deduplication. Although this remedy solution would reduce the deduplication rate, we still can perform out-of-line deduplication at a later time, so it is an acceptable tradeoff for retaining high performance.

3.5 Out-of-line Deduplication

As mentioned previously, CAFTL does not pursue a perfect in-line deduplication, and an internal routine is periodically launched to perform *out-of-line fingerprinting* and *out-of-line deduplication* during the device idle time.

Out-of-line fingerprinting is simple. We scan the metadata page array (Section 3.3.3) to find physical pages not yet fingerprinted. If one such a page is found, we read the page out, compute the fingerprint, and update its metadata. To avoid unnecessarily scanning the metadata of pages already fingerprinted, we use one bit in an entry of the metadata page array to denote if all of the fingerprints in the corresponding metadata page have already been computed, and we skip over such pages.

Out-of-line deduplication is more complicated due to the memory space constraint. We adopt a solution similar to the widely used *external merge sort* [39] in database systems. Supposing we have M fingerprints in total and the available memory space can accommodate N fingerprints, where $M > N$. We scan the metadata page array from the beginning, each time N fingerprints are loaded and sorted in memory, and temporarily stored in flash, then we load and sort the next N fingerprints, and so on. This process is repeated for K times ($K = \lceil \frac{M}{N} \rceil$) until all the fingerprints are processed. Then we can merge sort these K blocks of fingerprints in memory and identify the duplicate fingerprints.

Out-of-line fingerprinting and deduplication can be performed together with the GC process or independently. Since there is no harm in leaving duplicate or unfingerprinted pages in flash, these operations can be performed during idle period and immediately aborted upon incoming requests, and the perceivable performance impact to foreground jobs is minimal.

4 Performance Evaluation

4.1 Experimental Systems

We have implemented and evaluated our design of CAFTL based on a comprehensive trace-driven simulation. In this section we will introduce our simulator, trace collection, and system configurations.

4.1.1 SSD Simulator

CAFTL is a device-level design running in the SSD controller. We have implemented it in a sophisticated SSD simulator based on the Microsoft® Research SSD extension [5] for the *DiskSim* simulation environment [14]. This extension was also used in prior work [6].

The Microsoft extension is well modularized and implements the major components of FTL, such as the indirect mapping, garbage collection and wear-leveling policies, and others. Since the current version lacks an on-device buffer, which is becoming a standard component in recent generations of SSDs, we augmented the current implementation and included a shared buffer for handling incoming read and write requests. When a write request is received at the SSD, it is first buffered in the cache, and the SSD immediately reports completion to the host. Data processing and flash operations are conducted asynchronously in the background [16]. A read request returns back to the host once the data is loaded from flash into the buffer. We should note that this simulator follows a general FTL design [6], and the actual implementations of the SSD on the market can have other specific features.

4.1.2 SSD Configurations

Description	Configuration
Flash Page Size	4KB
Pages per Block	64
Blocks per Plane	2048
Planes per Package	8
# of Packages	10
Mapping policy	Full striping
Over-provisioning	15%
Garbage Collection Threshold	5%

Table 1: Configurations of the SSD simulator.

In our experiments, we use the default configurations from the SSD extension, unless denoted otherwise. Table 1 gives a list of the major config parameters.

Description	Latency
Flash Read/Write/Erase	25 µs/200µs/1.5ms
SHA-1 hashing (4KB)	47,548 cycles
CRC32 hashing (4KB)	4,120 cycles

Table 2: Latencies configured in the SSD simulator.

Table 2 gives the parameters of latencies used in our experiments. For the flash memory, we use the default latencies in our experiments. For the hashing latencies, we first cross compile the hash function code to the ARM® platform and run it on the *SimpleScalar-ARM* simulator [4] to extract the total number of cycles for executing a hash function. We assume a processor similar to ARM® Cortex R4 [8] on the device, which is specifically designed for high-performance embedded devices, including storage. Based on its datasheet, the ARM processor has a frequency from 304MHz to 934MHz [8], and we can estimate the latency for hashing a 4KB page by dividing the number of cycles by the processor frequency. It is also worth mentioning here that according to our communications with SSD manufacturer [3], high-frequency (600+ MHz) processors, such as the Cortex processor, are becoming increasingly normal in high-speed storage devices. Leveraging such abundant computing power on storage devices can be a research topic for further investigation.

4.1.3 Workloads and trace collection

We have selected 11 workloads from three representative categories and collected their data access traces.

- *Desktop* (d1,d2) – Typical office workloads, e.g. Internet surfing, emailing, word editing, etc. The workloads run for 12 and 19 hours, respectively, and feature irregular idle intervals and small reads and writes.
- *Hadoop* (h1-h7)– We execute seven TPC-H data warehouse queries (Query 1,6,11,14,15,16,20) with scale factor of 1 on a Hadoop distributed system platform [2]. These workloads run for 2-40 minutes and generate intensive large writes of temp data.
- *Transaction* (t1,t2) – We execute TPC-C workloads (1-3 warehouses, 10 terminals) for transaction processing on PostgreSQL 8.4.3 database system. The two workloads run for 30 minutes and 4 hours, respectively, and feature intensive write operations.

The traces are collected on a DELL® Dimension 3100 workstation with an Intel® Pentium™4 3.0GHz processor, a 3GB main memory, and a 160GB 7,200 RPM Seagate® hard disk drive. We use Ubuntu 9.10 with the Ext3 file system. We modified the Linux kernel 2.6.32 source code to intercept each I/O request and compute a SHA-1 hash value as a fingerprint for each 4KB page of the request. These fingerprints, together with other request information (e.g. offset, type), are transferred to another machine via *netconsole* [35]. This avoids the possible interference caused by tracing. The collected trace files are analyzed offline and used to drive the simulator for our experimental evaluation.

4.2 Effectiveness of Deduplication

CAFTL intends to remove duplicate writes and extend flash space. In this section, we perform two sets of experiments to show the effectiveness of deduplication in CAFTL. In both experiments, we use an SSD with a 934MHz processor and a 16MB buffer.

4.2.1 Removing duplicate writes

CAFTL identifies and removes duplicate writes via inline deduplication. Denoting the total number of pages requested to be written as n, and the total number of pages being actually written into flash medium as m, the *deduplication rate* is defined as $\frac{n-m}{n}$. Figure 8 shows the deduplication rate of the 11 workloads running on CAFTL. In this figure, *offline* refers to the optimal case, where the traces are examined and deduplicated offline. We also show CAFTL without sampling and with a sampling unit size of 128KB (32 pages), denoted as *no-sampling* and *128KB*, respectively.

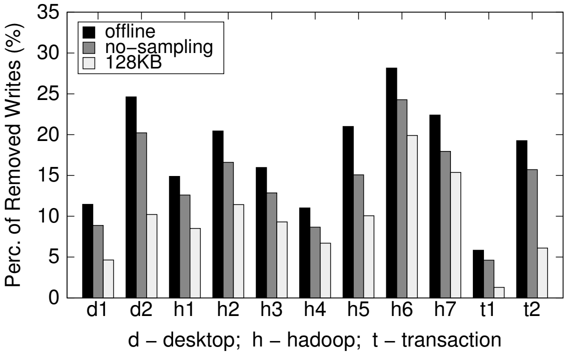


Figure 8: Perc. of removed duplicate writes.

As we see in Figure 8, duplication is highly workload dependent. Across the 11 workloads, the rate of duplicate writes in the workloads ranges from 5.8% (*t1*) to 28.1% (*h6*). CAFTL can achieve deduplication rates from 4.6% (*t1*) to 24.2% (*h6*) with no sampling. Compared with the optimal case (*offline*), CAFTL identifies up to 86.2% of the duplicate writes in *offline*. We also can see that with a larger sampling unit (128KB), CAFTL achieves a lower but reasonable deduplication rate. In Section 4.4.1, we will give more detailed analysis on the effect of sampling unit sizes.

4.2.2 Extending flash space

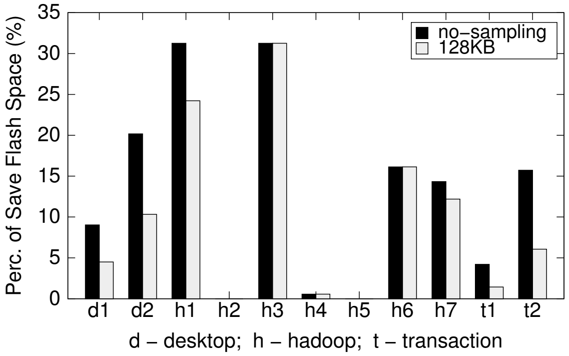


Figure 9: Perc. of extended flash space.

Besides directly removing duplicate writes to the flash memory, CAFTL also reduces the amount of occupied flash memory space and increases the number of available clean erase blocks for garbage collection and wear-leveling. Figure 9 shows the percentage of extended flash space in units of erase blocks, compared to the baseline case (without CAFTL). We show CAFTL without sampling (*no-sampling*) and with sampling (*128KB*).

As shown in Figure 9, CAFTL can save up to 31.2% (*h1*) of the occupied flash blocks for the 11 workloads. The worst cases are *h2* and *h5*, in which no space saving is observed. This is because the two workloads are relatively smaller, the total number of occupied erase blocks is only 176. Although the number of pages being written is reduced by 16.6% (*h2*) and 15% (*h5*), the saved space in units of erase blocks is very small.

4.3 Performance Impact

To make CAFTL truly effective in practice, we must retain high performance and minimize negative impact. Here we study three key factors affecting performance, *cache size*, *hashing speed*, and *fingerprint searching*. The acceleration methods are not applied in experiments.

4.3.1 Cache size

In Figure 10, we show the percentage of the increase of average read/write latencies with various cache sizes (2MB to 16MB). We compare CAFTL with the baseline case (without CAFTL). In the experiments, we configure an SSD with a 934MHz processor. We can see that with a small cache space (2MB), the read and write latencies can increase by a factor of up to 34% (*t1*). With a moderate cache size (8MB), the latency increases are reduced to less than 4.5%. With a 16MB cache, a rather standard size, the latency increases become negligible (less than 0.5%). For some workloads (*d2, h3, h5, h7, t1, t2*), we can even see a slight performance improvement (0.2-0.5%), because CAFTL removes unnecessary writes, which reduces the probability of being blocked by an in-progress flash write operation. In this case we see a negative performance impact with a small cache space, and we will show how to mitigate such a problem through our acceleration methods in Section 4.4.

4.3.2 Hashing speed

Computing fingerprints is time consuming and affects access performance. The hashing speed depends on the capability of processors. Using a more powerful processor can effectively reduce the latency for digesting pages and generating fingerprints. To study the performance impact caused by hashing speed, we vary the processor frequency from 304MHz to 934MHz, based on the Cortex datasheet [8]. We configure an SSD with a 16MB cache space and show the increase of read latencies compared to the baseline case (without CAFTL) in Figure 11. We did not observe an increase of write latencies, since most writes are absorbed in the buffer.

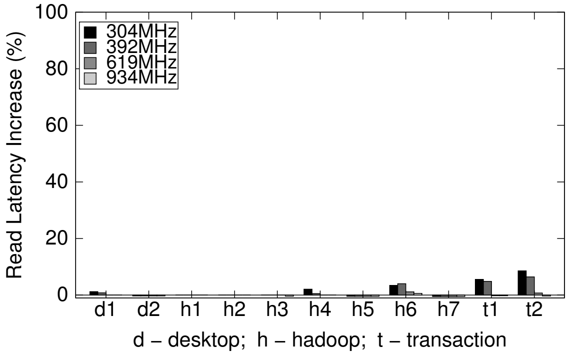


Figure 11: Perf. impact of hashing Speeds.

In Figure 11, we can see that most workloads are insensitive to hashing speed. With a 304MHz processor, the performance overhead is less than 8.5% (*t2*), which has more intensive larger writes. At 934MHz, the performance overhead is merely observable (up to 0.5%). There are two reasons. First, the 16MB on-device buffer absorbs most incoming writes and provides a sufficient space for accommodating incoming reads. Second, the incoming read requests are given a higher priority than writes, which reduces noticeable delays in the critical path. These optimizations make reads insensitive to hashing speed and reduces noticeable latencies. Also

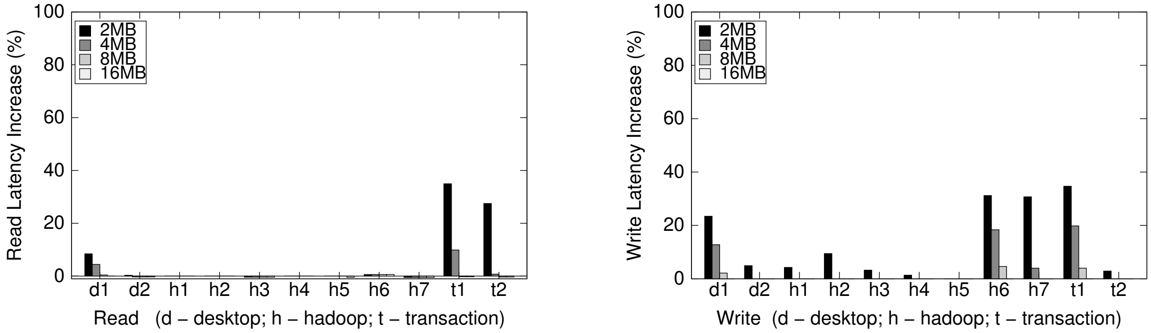


Figure 10: Performance impact of cache sizes (2-16MB).

note that if a dedicated hashing engine is used on the device, the hashing latency could be further reduced.

4.3.3 Fingerprint searching

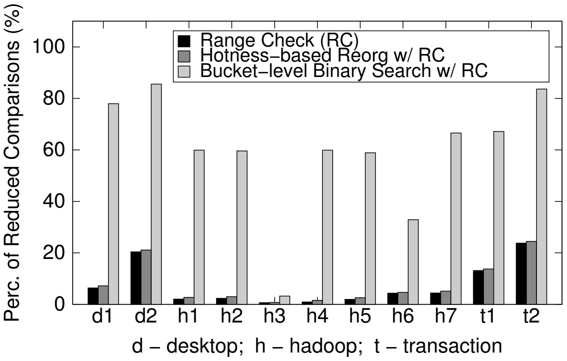


Figure 12: Optimizations on fingerprint searching.

We have proposed three techniques to accelerate fingerprint searching. Figure 12 shows the percentage of reduced fingerprint comparisons compared with the baseline case. We configure the fingerprint store with 256 segments to hold the fingerprints for each workload. We can see that using *Range Check* can effectively reduce the comparisons of fingerprints by up to 23.7% (*t2*). However, *Hotness-based Reorganization* can provide little further improvement (less than 1%), because it essentially accelerates lookups for fingerprints that are duplicated, which is relatively an uncommon case. As expected, *Bucket-level Binary Search* can significantly reduce the average number of comparisons for each lookup. In *d2*, for example, *Bucket-level Binary Search* can effectively reduce the average number of comparisons by a factor of 85.5%. Thus we would suggest applying *Bucket-level Binary Search* and *Range Check* to speed up fingerprint lookups.

4.4 Acceleration Methods

With a small on-device buffer, the high computational latency caused by hashing could be significant and perceived by the users. We have developed three techniques to accelerate fingerprinting. In this section, we will show the effectiveness of each individual technique and then show the effects in aggregate. We configure an SSD with a 934MHz processor and a small 2MB buffer.

4.4.1 Sampling

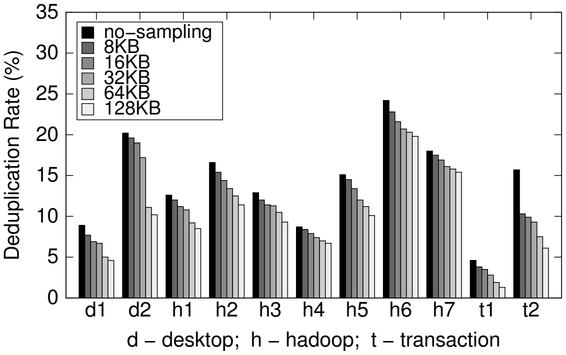


Figure 14: Dedup. with Sampling

As shown in Figure 13 and Figure 14, sampling can significantly improve performance. With the increase of sampling unit size, fewer fingerprints need to be calculated, which translates into a manifold reduction of observed read and write latencies. For example, *h7* achieves a speedup by a factor of 94.1 times for reads and 3.5 times for writes, because of the significantly reduced waiting time for the buffer. Meanwhile, the deduplication rate is only reduced from 18% to 15.4%. Considering such a significant speedup, the minor loss of deduplication rate is acceptable. The maximum speedup, 110.6 times (read), is observed in *t1*, and its deduplication rate drops from 4.6% to 1.3%. This is mostly because for workloads with low duplication rate, the probability of sampling right pages is also relatively low.

4.4.2 Light-weight pre-hashing

Light-weight pre-hashing uses a fast CRC32 hash function to filter most unlikely-to-be-duplicated pages before performing high-cost fingerprinting. Figure 15 shows the speedup of reads and writes by using CRC32 for pre-hashing, compared with CAFTL without pre-hashing. Only pre-hashing is enabled here. We can see that in the best case (*t1*), pre-hashing can reduce the latencies by a factor of up to 148.3 times for reads and 3.9 times for writes. This is because, as mentioned previously,

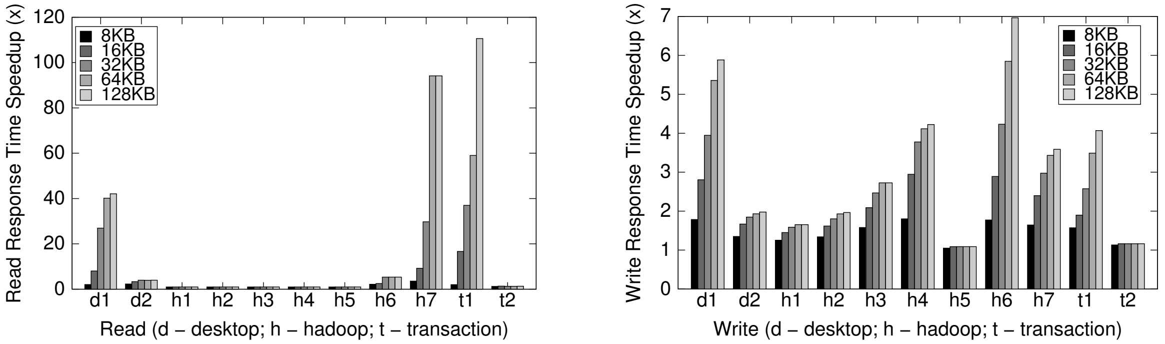


Figure 13: Performance speedup with Sampling (unit size: 8-128KB).

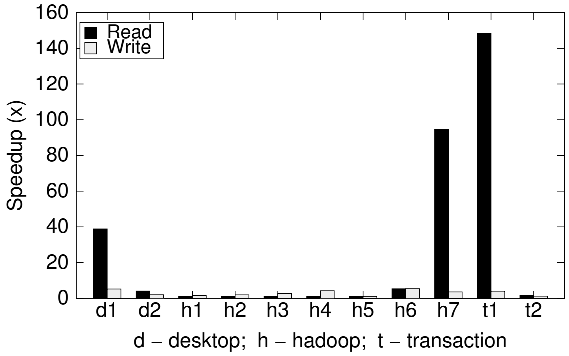


Figure 15: Speedup with pre-hashing.

this workload is write intensive and has a long waiting queue, which makes the queuing effect particularly significant. Similar to sampling, writes receive relatively smaller benefit, because the buffer absorbs the writes with low latency and diminishes the effect of speeding up writes. Meanwhile, we also found negligible difference in deduplication rates, which is consistent with our analysis shown in Figure 7.

4.4.3 Dynamic switch

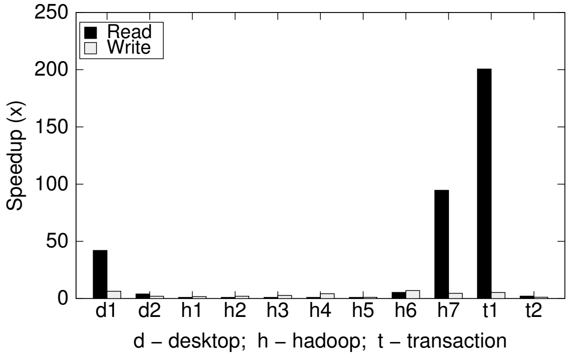


Figure 16: Speedup with dynamic switch.

CAFTL also provides *dynamic switch* to dynamically turn on/off the in-line deduplication, depending on the usage of the on-device buffer. We configure the high watermark as 95% (off) and the low watermark as 50% (on). Figure 16 shows the speedup of reads and writes in the workloads. Again, *t1* receives the most significant performance speedup by a factor of 200.6 times. Some workloads (*h1*-*h5*) receive no benefits, because they are less I/O intensive. For the other workloads, we can observe a speedup of 2.1 times to 94.6 times.

4.4.4 Putting it all together

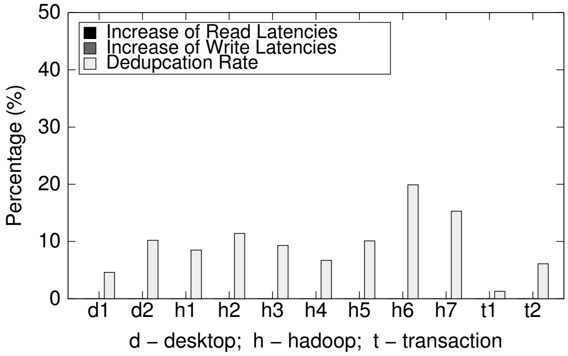


Figure 17: Three acceleration tech. combined

In Figure 17, we enable all the three acceleration techniques and show the increase of read and write latencies, compared with the baseline case (without CAFTL), and the corresponding deduplication rate. We can see that by combining all the three techniques, we can almost completely remove the performance overhead with only a 2MB on-device buffer. In the meantime, we can achieve a deduplication rate of up to 19.9%.

5 Other Related Work

Flash memory based SSDs have received a lot of interest in both academia and industry. There is a large body of research work on flash memory and SSDs (e.g. [6,9,13,15–18,20,23,26–29,31,34,37,38,42,44]). Concerning lifespan issues, most early work focuses on designing garbage collection and wear-leveling policies. A survey [21] summarizes these techniques. Here we only present the papers most related to this work.

Recently Grupp et al. [22] have presented an empirical study on the performance, power, and reliability of flash memories. Their results show that flash memories, particularly MLC devices, exhibit significant error rates after or even before reaching the rated lifetime, which makes using high density SSDs in commercial systems a difficult choice. Another report [13] has studied the write endurance of USB flash drives with a more optimistic conclusion that the endurance of flash memory chips is better than expected, but whole-device endurance is closely related to the FTL designs. A modeling based study on the endurance issues has also been presented in [33]. These studies provide much needed information about the lifespan of flash memory and small-size flash devices. However, so far the endurance of state-of-the-art SSDs has not yet been proven in the field [10].

Early studies on SSDs mainly focus on performance. Some recent studies have begun to look at reliability issues. Differential RAID [9] tries to improve reliability of an SSD-based RAID storage by distributing parity unevenly across SSDs to reduce the probability of correlated multi-device failure. Griffin [42] extends SSD lifetime by maintaining a log-structured HDD cache and migrating cached data periodically. A recent work [36] considers write cycles in addition to storage space as a constrained resource in depletable storage systems and suggests attribute depletion to users in systems like cloud computing. ChunkStash [19] uses flash memory to speed up index lookups for inline storage deduplication. Another work [43] proposes to integrate phase change memory into SSDs to improve the performance, energy consumption, and also lifetime. Our study has made its unique contributions to enhancing the lifespan of SSDs by removing duplicate writes and coalescing redundant data at the device level, as a more general solution.

6 Conclusion and Discussions

Enhancing the SSD lifespan is crucial to a wide deployment of SSDs in commercial systems. In this paper, we have proposed a solution, called CAFTL, and shown that by removing duplicate writes and coalescing redundant data, we can effectively enhance the lifespan of SSDs while retaining high data access performance.

A potential concern about CAFTL is the volatility of the on-device RAM buffer – the buffered data could be lost upon power failure. However, this concern is not new to SSDs. A hard disk drive also has an on-device buffer, but it provides users an option (e.g. using *sdparm* tool) to flexibly enable/disable the buffer on their needs. Similarly, if needed, the users can choose to disable the in-line deduplication and the buffer in an SSD, and the out-of-line deduplication can still be effective.

Although we have striven to minimize memory usage, CAFTL demands more space for storing fingerprints and the secondary mapping table, compared with traditional FTLs. According to our communications with SSD manufacturer [3], memory actually only accounts for a small percentage of the total production cost, and the most expensive component is flash memory. Thus we consider this tradeoff is worthwhile to extend available flash space, and SSD lifespan. If budget allows, we would suggest maintaining the fingerprint store fully in memory, which not only improves deduplication rate but also simplifies designs.

Further improvements are also possible. One is to relax the stringent "one-time programming" requirement. According to the specification, each flash page in a clean erase block should be programmed (written) only once. In practice, flash chips can allow multiple programs to a page and the risk of "program disturb" is fairly low [7]. We can leverage this feature to simplify many designs. For example, we can write multiple versions of LBA/VBA and fingerprints into the spare area of a physical page, which can largely remove the need for metadata pages. Another consideration is to integrate a byte-addressable persistent memory (e.g. PCM) into the SSDs to maintain the metadata, which can remove much design complexity. We are also considering the addition of on-line compression into SSDs to better utilize the high-speed processor on the device. This can further extend available flash space but may require more changes to the FTL design, which will be our future work.

As SSD technology becomes increasingly mature and delivers satisfactory performance, we believe, the endurance issue of SSDs, particularly high-density MLC SSDs, opens many new research opportunities and should receive more attention from researchers.

Acknowledgments

We are grateful to our shepherd Dr. Christos Karamanolis from VMware® and the anonymous reviewers for their constructive comments. We also thank our colleague Bill Bynum for reading this paper and his suggestions. This research was partially supported by the NSF under grants CCF-0620152, CCF-072380, and CCF-0913150.

References

[1] FIPS 180-1, Secure Hash Standard, April 1995.

[2] Hadoop. http://hadoop.apache.org/, 2010.

[3] Personal communications with an SSD architect, 2010.

[4] SimpleScalar 4.0. http://www.simplescalar.com/v4test.html, 2010.

[5] SSD extension for DiskSim simulation environment. http://research.microsoft.com/en-us/downloads/b41019e2-1d2b-44d8-b512-ba35ab814cd4/, 2010.

[6] AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J. D., MANASSE, M., AND PANIGRAHY, R. Design tradeoffs for SSD performance. In *Proceedings of USENIX'08* (Boston, MA, June 2008).

[7] ANDERSEN, D. G., AND SWANSON, S. Rethinking flash in the data center. In *IEEE Micro* (July/Aug 2010).

[8] ARM. Cortex R4. http://www.arm.com/products/processors/cortex-r/cortex-r4.php, 2010.

[9] BALAKRISHNAN, M., KADAV, A., PRABHAKARAN, V., AND MALKHI, D. Differential RAID: Rethinking RAID for SSD Reliability. In *Proceedings of EuroSys'10* (Paris, France, April 2010).

[10] BARROSO, L. A. Warehouse-scale computing. In *Keynote in the SIGMOD'10 conference* (Indianapolis, IN, June 2010).

[11] BHAGWAT, D., ESHGHI, K., LONG, D. D. E., AND LILLIBRIDGE, M. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Proceedings of MASCOTS'09* (London, UK, September 2009).

[12] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. In *Communications of the ACM* (1970), vol. 13(7), pp. 422–426.

[13] BOBOILA, S., AND DESNOYERS, P. Write endurance in flash drives: Measurements and analysis. In *Proceedings of FAST'10* (San Jose, CA, February 2010).

[14] BUCY, J., SCHINDLER, J., SCHLOSSER, S., AND GANGER, G. DiskSim 4.0. http://www.pdl.cmu.edu/DiskSim, 2010.

[15] CHEN, F., JIANG, S., AND ZHANG, X. SmartSaver: Turning flash drive into a disk energy saver for mobile computers. In *Proceedings of ISLPED'06* (Tegernsee, Germany, October 2006).

[16] CHEN, F., KOUFATY, D. A., AND ZHANG, X. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proceedings of SIGMETRICS/Performance'09* (Seattle, WA, June 2009).

[17] CHEN, F., LEE, R., AND ZHANG, X. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Proceedings of HPCA'11* (San Antonio, TX, Feb 2011).

[18] CHEN, S. FlashLogging: Exploiting flash devices for synchronous logging performance. In *Proceedings of SIGMOD'09* (Providence, RI, June 2009).

[19] DEBNATH, B., SENGUPTA, S., AND LI, J. ChunkStash: Speeding up inline storage deduplication using flash memory. In *Proceedings of USENIX'10* (Boston, MA, June 2010).

[20] DIRIK, C., AND JACOB, B. The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device, architecture, and system organization. In *Proceedings of ISCA'09* (Austin, TX, June 2009).

[21] GAL, E., AND TOLEDO, S. Algorithms and data structures for flash memories. In *ACM Computing Survey'05* (2005), vol. 37(2), pp. 138–163.

[22] GRUPP, L. M., CAULFIELD, A. M., COBURN, J., SWANSON, S., YAAKOBI, E., SIEGEL, P. H., AND WOLF, J. K. Characterizing flash memory: Anomalies, observations, and applications. In *Proceedings of MICRO'09* (New York, NY, December 2009).

[23] GUPTA, A., KIM, Y., AND URGAONKAR, B. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceedings of ASPLOS'09* (Washington, D.C., March 2009).

[24] GUPTA, D., LEE, S., VRABLE, M., SAVAGE, S., SNOEREN, A. C., VARGHESE, G., VOELKER, G. M., AND VAHDAT, A. Difference Engine: Harnessing memory redundancy in virtual machines. In *Proceedings of OSDI'08* (San Diego, CA, 2008).

[25] INTEL. Intel X25-E extreme SATA solid-state drive. http://www.intel.com/design/flash/nand/extreme, 2008.

[26] JOSEPHSON, W. K., BONGO, L. A., FLYNN, D., AND LI, K. DFS: A file system for virtualized flash storage. In *Proceedings of FAST'10* (San Jose, CA, February 2010).

[27] KAWAGUCHI, A., NISHIOKA, S., AND MOTODA, H. A flash-memory based file system. In *Proceedings of USENIX Winter* (New Orleans, LA, Jan 1995), pp. 155–164.

[28] KIM, H., AND AHN, S. BPLRU: A buffer management scheme for improving random writes in flash storage. In *Proceedings of FAST'08* (San Jose, CA, February 2008).

[29] LEE, S., AND MOON, B. Design of flash-based DBMS: An in-page logging approach. In *Proceedings of SIGMOD'07* (Beijing, China, June 2007).

[30] LILLIBRIDGE, M., ESHGHI, K., BHAGWAT, D., DEOLALIKAR, V., TREZISE, G., AND CAMBLE, P. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *Proceedings of FAST'09* (San Jose, CA, 2009).

[31] MAKATOS, T., KLONATOS, Y., MARAZAKIS, M., FLOURIS, M. D., AND BILAS, A. Using transparent compression to improve SSD-based I/O caches. In *Proceedings of EuroSys'10* (Paris, France, April 2010).

[32] MENEZES, A. J., V. OORSCHOT, P. C., AND VANSTONE, S. A. Handbook of applied cryptography. In *CRC Press* (1996).

[33] MOHAN, V., SIDDIQUA, T., GURUMURTHI, S., AND STAN, M. R. How I learned to stop worrying and love flash endurance. In *Proceedings of HotStorage'10* (Boston, MA, June 2010).

[34] NARAYANAN, D., THERESKA, E., DONNELLY, A., ELNIKETY, S., AND ROWSTRON, A. Migrating enterprise storage to SSDs: analysis of tradeoffs. In *Proceedings of EuroSys'09* (Nuremberg, Germany, March 2009).

[35] NETCONSOLE. http://www.kernel.org/doc/Documentation/ networking/netconsole.txt, 2010.

[36] PRABHAKARAN, V., BALAKRISHNAN, M., DAVIS, J. D., AND WOBBER, T. Depletable storage systems. In *Proceedings of HotStorage'10* (Boston, MA, June 2010).

[37] PRABHAKARAN, V., RODEHEFFEER, T. L., AND ZHOU, L. Transactional flash. In *Proceedings of OSDI'08* (San Diego, CA, December 2008).

[38] PRITCHETT, T., AND THOTTETHODI, M. SieveStore: A highly-selective, ensemble-level disk cache for cost-performance. In *Proceedings of ISCA'10* (Saint-Malo, France, June 2010).

[39] RAMAKRISHNAN, R., AND GEHRKE, J. Database managment systems. McGraw-Hill, 2030.

[40] RIVEST, R. The MD5 message-digest algorithm. http://www.ietf.org/rfc/rfc1321.txt, April 1992.

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

[42] SOUNDARARAJAN, G., PRABHAKARAN, V., BALAKRISHNAN, M., AND WOBBER, T. Extending SSD lifetimes with disk-based write caches. In *Proceedings of FAST'10* (San Jose, CA, February 2010).

[43] SUN, G., JOO, Y., CHEN, Y., NIU, D., XIE, Y., CHEN, Y., AND LI, H. A hybrid solid-state storage architecture for the performance, energy consumption, and lifetime improvement. In *Proceedings of HPCA'10* (Bangalore, India, Jan 2010).

[44] TSIROGIANNIS, D., HARIZOPOULOS, S., AND SHAH, M. A. Query processing techniques for solid state drives. In *Proceedings of SIGMOD'09* (Providence, RI, June 2009).

[45] UNGUREANU, C., ATKIN, B., ARANYA, A., GOKHALE, S., RAGO, S., CALKOWSKI, G., DUBNICKI, C., AND BOHRA, A. HydraFS: A high-throughput file system for the HYDRAstor content-addressable storage system. In *Proceedings of FAST'10* (San Jose, CA, 2010).

[46] WIKIPEDIA. Battery or supercap. http://en.wikipedia.org/wiki/ Solid-state-drive#Battery_or_SuperCap, 2010.

[47] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of FAST'08* (San Jose, CA, 2008).

Leveraging Value Locality in Optimizing NAND Flash-based SSDs

Aayush Gupta, Raghav Pisolkar, Bhuvan Urgaonkar, and Anand Sivasubramaniam
{axg354,rvp116,bhuvan,anand}@cse.psu.edu
Department of Computer Science and Engineering
The Pennsylvania State University, University Park 16802, PA

Abstract: NAND flash-based solid-state drives (SSDs) are increasingly being deployed in storage systems at different levels such as buffer-caches and even secondary storage. However, the poor reliability and performance offered by these SSDs for write-intensive workloads continues to be their key shortcoming. Several solutions based on traditionally popular notions of temporal and spatial locality help reduce write traffic for SSDs. However, another form of locality - *value locality* - has remained completely unexplored. Value locality implies that certain data items (i.e., "values," not just logical addresses) are likely to be accessed preferentially. Given evidence for the presence of significant value locality in real-world workloads, we design CA-SSD which employs content-addressable storage (CAS) to exploit such locality. Our CA-SSD design employs enhancements primarily in the flash translation layer (FTL) with minimal additional hardware, suggesting its feasibility. Using three real-world workloads with content information, we devise statistical characterizations of two aspects of value locality - value popularity and temporal value locality - that form the foundation of CA-SSD. We observe that CA-SSD is able to reduce average response times by about 59-84% compared to traditional SSDs. Even for workloads with little or no value locality, CA-SSD continues to offer comparable performance to a traditional SSD. Our findings advocate adoption of CAS in SSDs, paving the way for a new generation of these devices.

1 Introduction and Motivation

NAND flash-based SSDs offer several advantages over magnetic hard disks: lower access latencies, lower power consumption, lack of noise, and higher robustness to vibrations and temperature. Several researchers have explored the performance benefits of employing these SSDs, either as complete replacements for magnetic drives or in supplementary roles (e.g., caches) [23]. Whereas a number of other non-volatile memory technologies - phase-change, ferroelectric, and magnetic RAM - exist at different levels of maturity and offer similar benefits, cost/feasibility projections suggest that NAND flash (simply flash, henceforth) is likely to be at the forefront of these significant changes in storage for the next decade [17]. Another trend from EMC suggests that SSD prices will continue to fall to the extent of becoming cheaper than 15K RPM HDDs by 2017 [7]. Thus, exploring ways to further improve flash technology and its use in designing better storage systems will continue to be worthwhile pursuits in the coming years.

Flash is a unique memory technology due to the sensitivity of its reliability and performance to write traffic. A flash page (the granularity of reads/writes) must be erased before it may be written. Erases occur at the granularity of blocks which contain multiple pages. Furthermore, blocks become unreliable after 5K-100K erase operations [38, 39, 37]. This erase-before-write property of flash necessitates out-of-place updates to prevent the relatively high latency of erases from affecting the performance of writes. These out-of-place updates create invalid pages that contain older versions of data requiring garbage collection. This further exacerbates the reliability/performance concerns by introducing additional write operations. Techniques that reduce the number of writes to SSDs are, therefore, desirable and have received a lot of attention. Existing approaches for write reduction have relied on exploiting the presence of (i) temporal locality (e.g., buffering writes within file system/SSD/other media to eliminate duplicate writes to flash [24, 46, 45]), and/or (ii) spatial locality (e.g., coalescing multiple sub-page writes into fewer page writes [30]) within workloads. However, there is yet another dimension of locality - *value locality* - that has remained unexplored for flash SSDs. The presence of value locality in a workload means that it preferentially accesses certain content (i.e., values) over others. This property facilitates data de-duplication (storing only one copy of each unique value), which is especially attractive for SSDs as it naturally offers the write reduction that these devices can benefit from: a SSD employing such data de-duplication need not do an additional write of a value that it has already stored. This benefit applies even if the two writes belong to entirely different logical addresses and even in the absence of any temporal/spatial correlation between these two writes. Data de-duplication can also reduce read traffic, with additional performance benefits.

Content addressable storage (CAS) is a popular deduplication technique which operates on data by dividing it into non-intersecting *chunks*, and employing a cryptographic hash to represent each chunk. By storing only unique hashes (and their corresponding data chunks), duplicate chunks in data are removed. Hashing can result in collisions where different data blocks can be mapped to the same value. However, it has been shown that such collisions are practically unlikely, with probabilities in the range $10^{-9} - 10^{-17}$ [40, 42] for MD5 and SHA-1. Additionally, techniques to further reduce this probability to as low as 10^{-46} have been shown to be feasible [40, 43]. Thus, consistent with most CAS research [12, 42, 35], we also assume hash functions to be collision-resistant. CAS has been extensively used in archival and backup systems [42, 43, 14], but its benefits specific to SSDs have not been explored. Whereas SSDs could benefit from existing host-level (e.g., file system [47]) implementations of CAS, thereby reducing I/O traffic, there is significant motivation to realize this functionality within the device itself. It allows incorporation of value locality without requiring any modifications to the upper layers (filesystem, block layer etc.), thus allowing quick adoption in existing systems. Several SSD optimizations that rely upon information about flash data layout are better implemented within the SSD. For example, garbage collection efficiency can be improved by using data placement policies which reduce overheads of copying valid pages. Also, scalability of a CAS-based scheme crucially depends on its ability to carry out fast calculations/look-ups of hashes. This can be achieved by using dedicated hardware such as that increasingly available in SSDs (e.g., those with Full Disk Encryption capabilities [5, 44, 41]), relieving the host of these computational overheads.

Key Choices and Challenges: A number of interesting design choices and challenges arise when designing a SSD that employs CAS for its internal data management. First, in order to maintain compatibility with existing storage software, we choose that our SSD continue to expose its existing block interface. Modifications to the SSD interface such as nameless writes [11] can potentially benefit CA-SSD but require changes to the upper layers. Second, employing CAS necessitates several enhancements to the data structures maintained by our SSD's flash translation layer (FTL). This increased "meta-data" puts additional pressure on the scarce on-SSD RAM and must be managed carefully. Third, data de-duplication renders ineffective existing mechanisms employed by the FTL to recover its meta-data after power failures. Existing FTLs store information about the logical address (LPN) stored on a flash page in a special region called the out-of-band area (OOB) within the page itself. Due to de-duplication with CAS, a given page may correspond to multiple LPNs (different LPNs may contain the same content) , and thus, its OOB area cannot be used as before. Fourth, with CAS the notion of when a page becomes invalid changes - a page should now be invalidated only when all the LPNs having that content have written a "different content" - implying a re-consideration of the design of the garbage collector. Finally, whereas we design our SSD to exploit value locality whenever present, we would like it not to exhibit degraded performance or reliability than a state-of-the-art SSD in the absence of such locality.

Research Contributions: We make the following contributions in this paper.

- We propose CA-SSD, a flash solid-state drive that employs CAS for internal data management and addresses all the concerns outlined above. We demonstrate how CA-SSD functionality can be achieved mostly by modifying the FTL and with minimal support in the form of additional hardware compared to traditional SSDs. This additional hardware is similar to that already present in many state-of-the-art SSDs.
- We identify and characterize salient aspects of value locality- value popularity and temporal value locality and design CA-SSD algorithms to exploit them.
- Using three real-world workloads with content information, we evaluate the efficacy of CA-SSD by simulations. We observe that CA-SSD is able to reduce the average response times by about 59–84% for these workloads. Additionally, from 10 real-world traces, we synthesize workloads with different degrees of value locality. We find that CA-SSD consistently outperforms traditional SSD with even small degrees of locality and offers comparable performance when there is little or no value locality.

The rest of this paper is organized as follows. In Section 2 we provide an overview of the design of our CA-SSD comparing it to traditional SSDs. We discuss key aspects of value locality that affect CA-SSD design in Section 3. We design CA-SSD using insights gained in Section 4 and evaluate it in Section 5. Finally, we present related work in Section 6 and conclude in Section 7.

2 Overview of Our CA-SSD

Type	Data Unit			Access Time			Lifetime
	Page (Bytes)		Block	Read	Write	Erase	Write/Erase
	Data	OOB	(Bytes)	(us)	(us)	(ms)	(cycles)
SLC1	2048	64	128K+4K	25	200	1.5	100K
SLC2	4096	128	256K+8K	25	500	1.5	100K
MLC	4096	224	512K+28K	60	800	2.5	5K

Table 1: SLC & MLC NAND Flash characteristics [38, 39, 37]. SLC1/SLC2 represent SLC SSDs with different page sizes. Read/write latencies are at the granularity of pages while erase latencies are for blocks.

In this section, we describe how a flash-based SSD works and provide an overview of the changes to implement our CA-SSD.

2.1 Flash Solid-State Drives: A Primer

Figure 1(a) presents the key components of a traditional NAND flash-based SSD. In addition to the read and write operations which are performed at the granularity of a *page*, flash also provides an *erase* operation which is performed at the granularity of a *block* (composed of 64-128 pages). The coarser spatial granularity of erases makes them significantly slower than reads/writes. Furthermore, there is an asymmetry in read and write latencies, with writes being slower than reads. Blocks are further arranged in *planes* which can allow simultaneous operations through multi-plane commands thus improving performance [10]. In this paper, we only consider a single plane and our ideas and results apply readily to multiple planes. A page must first be erased before it can be written. The erase-before-write property of flash memory necessitates out-of-place updates to prevent the relatively high latency of erases from affecting the performance of updates. These out-of-place updates result in invalidation of older versions of pages requiring Garbage Collection (GC) to reclaim certain invalid pages in order to create room for newer writes. At a high level, GC operates by erasing certain blocks after relocating any valid pages within them to new pages. A final characteristic concerns the lifetime of flash memory, which is limited by the number of erase operations on its cells. Each block typically has a lifetime of 5K(MLC) or 100K(SLC) erase operations. Wear leveling (WL) techniques [20, 22, 32] are employed by the FTL to maintain similar lifetime for all the blocks. Table 1 presents representative values for the operational latencies, page/block sizes, and lifetime for two main flash technologies (SLC and MLC) [38, 39, 37]. We consider SLC-based flash in this work, although our ideas apply equally to MLC.

The Flash Translation Layer (FTL) is a software layer that helps in emulating an SSD as a block device by hiding the erase-before-write characteristics of flash memory. The FTL consists of three main logical components: (i) a Mapping Unit that performs data placement and translation of logical-physical addresses, (ii) the GC, and (iii) the WL. A key data structure maintained by the FTL is a *Mapping Table* which stores address translations. Upon receiving a write/update request for a logical page the FTL: (i) chooses an erased physical page where it writes this data, (ii) invalidates the previous version (if any) of the page in question, and (iii) updates its mapping table to reflect this change. The Mapping Table is typically stored on SSD's RAM to allow fast translation[1].

2.2 SSD Enhancements for CAS

In Figure 1(b), we present the additional components/functionality (compared to a traditional drive) required by CA-SSD. For both devices, we also show the steps involved in processing requests coming from the block device driver to help understand the difference in their operation. We refer to the FTL in CA-SSD as *CA-FTL*. Read requests are handled identically in both the SSDs and so we only focus on write requests. Whereas a traditional SSD requires all writes to be sent to physical pages, CA-SSD returns a write request without requiring flash page writes if hashes, representing their content, are found in RAM. We require four key enhancements to a traditional SSD to achieve this functionality.

(i) *Hashing Unit*: CA-FTL requires the ability to compute/compare content hashes such that these operations only degrade the CA-SSD performance to a negligible (or tolerable) extent. To ensure this, we propose to employ a dedicated co-processor to implement our hashing unit. Recently, manufacturers like OCZ [5], Samsung [44] and pureSilicon [41] have developed high performance SSDs with on-board cryptographic processors, suggesting that the desired fast hashing is feasible.

(ii) *Additional Meta-data*: Mapping Unit must maintain additional data structures for CAS that puts additional pressure on the *on-SSD RAM*. These structures represent CA-FTL's *meta-data* (to be distinguished from the meta-data for software such as the file system) and the portion of on-SSD RAM used for storing it is referred as the *meta-data cache*. We describe these data structures and space-efficient ways of managing them in Section 4.1.

(iii) *Persistent Meta-data Store*: Our CA-SSD design necessitates a re-consideration of the mechanism for recovering the contents of the meta-data cache after a power failure. When writing a physical page (PPN), a traditional FTL also stores the logical page number (LPN) in a special-purpose part of the PPN called the

[1] An SSD typically has a small SRAM and a larger DRAM cache whose size is in the range 64-512 MB for an SSD with capacity 256-1024 GB [1, 6]. We ignore this distinction in our discussion.

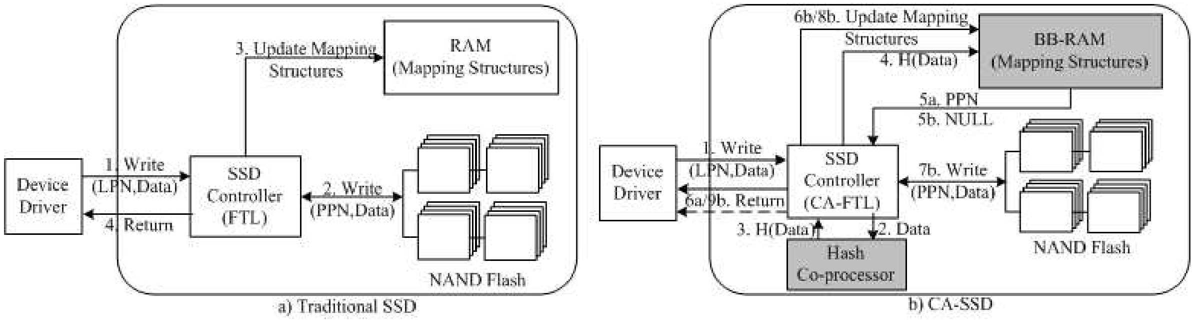


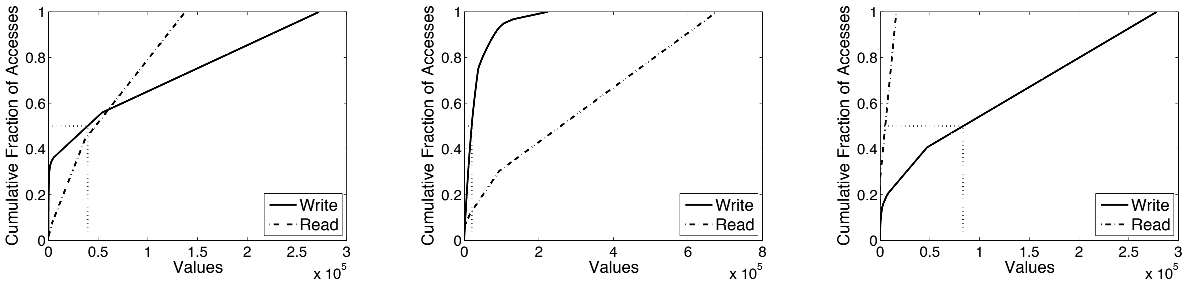
Figure 1: Components of a CA-SSD compared to traditional SSD. CA-SSD has two new hardware elements: (i) a hashing co-processor and (ii) a battery-backed RAM (BB-RAM). Furthermore, CA-SSD stores hashes instead of LPN in the page OOB area. Also shown is a comparison of how writes are handled in the two devices. (a) Traditional SSD: (1-2) On receiving a write request from device driver, SSD controller issues a flash page write. (3-4) On completion, the Mapping Table in the volatile RAM is updated and driver is notified of request completion. (b) CA-SSD: (1-2) On receiving a write request, the SSD controller sends the content to the hash co-processor for hash computation. (3-4) The returned hash is then looked up in the Mapping Table in the BB-RAM. (5-6(a)) On a hit, the mapping structures are updated and the request completes. (5-9(b)) On a miss, a flash page write is performed, mapping structures are updated and the request is completed.

out-of-band (OOB) area, which is typically 64-224 B in size. After a power failure, these entries in the OOB are used to reconstruct the LPN-to-PPN mappings. In CA-FTL, multiple LPNs may contain the same value and hence correspond to the same PPN. The OOB area may not have enough room for all these LPNs. Furthermore, a value can be associated with a changing set of LPNs over its lifetime, requiring multiple writes to the same OOB area, with corresponding erase/copying operations. We address this difficulty by requiring that CA-FTL's Mapping Table be kept in a fast persistent storage in the first place, without any need to store a copy on flash. Storing a copy on flash would result in large number of meta-data writes on flash increasing the number of flash page writes. An alternative approach could be to perform periodic check-pointing of Mapping Table instead of immediate writes on flash to reduce the number of meta-data writes, thereby providing weaker guarantees on meta-data consistency. In order to provide consistency guarantees similar to existing SSDs without impacting the overall performance, we employ persistent battery-backed RAM. We indicate this as BB-RAM in Figure 1(b). Write caches based on such battery-backed DRAM are commonly used in RAID controllers [3]. Even SSD manufacturers have started providing battery-backed DRAM as a standard feature to deal with power failures [5, 4]. Such SSDs with both battery-backed DRAM as well as on-board cryptographic processors have similar performance and costs as compared to traditional SSDs [5] mitigating performance and cost concerns for CA-SSD. Recent work has considered employing other persistent media (e.g., PCM [45] and even hard disk [46]) for SSD write optimizations, and exploring such alternatives for CA-SSD meta-data cache is part of future work.

(iv) *Re-design of GC*: CAS results in a change to GC. In conventional FTLs, each update results in the invalidation of a page requiring an eventual erase operation. But *CA-FTL only needs to invalidate a page when no LPN points to the value in that page*. This redefines the way garbage is created and distributed in blocks impacting the efficiency of GC. We study these issues in Section 4.2. We do not modify WL policy in this work and assume CA-SSD continues to employ the default WL.

3 Value Locality Characterization

We describe two aspects of value locality (VL) that have performance/lifetime implications for a CA-SSD. We propose ways to express these aspects statistically and discuss their implications for possible improvements in CA-SSD. Throughout our discussion, we employ three workload traces [26] described in Table 2 to present examples of our VL characterization. *homes* represents a file server of the home directories of a research group in FIU's CIS department. A major source of content similarity in this workload can be attributed to work done by different members of the group on copies of same software codes, technical documents etc. present in their directories. *mail* has been collected from the e-mail server of the same department containing similar mailing-list emails and circulated attachments resulting in content similarity across user INBOXes. Finally, *web* is their Web server workload consisting of virtual machines hosting an online course management system



(a) web

(b) mail

(c) homes

Figure 2: Value popularity in real-world workloads (1 day traces). The x-axis consists of unique values sorted according to their read or write popularity. That is, a given point on the x-axis might correspond to different values for reads and writes. We also show the number of unique values that correspond to 50% of all write requests.

Workload	Size (GB)	% Writes	Req. (mill.)	Unique Request (%)		Seq. %
				Write	Read	
web	1.95	77.01	3.8	42.35	32.05	83.8
mail	4.22	77.32	3.6	7.83	80.85	94.7
homes	3.02	96.76	4.4	66.37	80.75	70.8

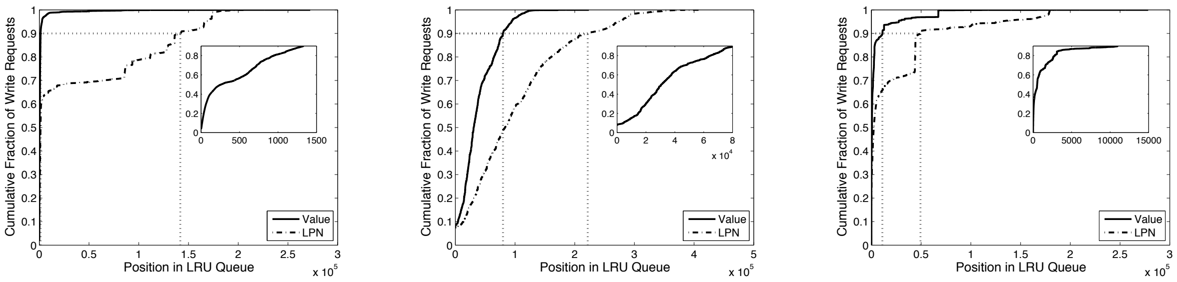
Table 2: Workload statistics. Workload duration varies from 1 day (*mail*) to 7 days (*web*,*homes*). Size represents the total number of unique LPNs accessed in the trace over the mentioned duration and hence represents a compacted trace without any intermediate non-accessed LPNs (The SSD size chosen for evaluation is 4GB for *homes* & *web* and 6GB for *mail*). The logical address space exposed to the file-system is much larger [26]. Unique Request denotes the fraction of write(read) requests which write(read) unique 4KB chunks. Requests are deemed sequential(seq.) if they access consecutive LPNs.

and email access portal. These workloads are primarily write-dominant, especially *homes*, which has about 97% write requests. Individual requests in these workloads are of size 4KB, along with a 16B hash(MD5) of the contents.

Value Popularity (VP): The most straightforward characterization of VL represents the popularity (number of occurrences) of each unique value, for both reads and writes separately. The VL for writes and reads have different implications for CA-SSD: whereas the former captures reduction in write traffic offered by caching the corresponding (value, LPN, physical page) information in the meta-data cache, the latter captures reduction in reads due to caching the corresponding content in the content cache. Table 2 shows the high VP exhibited by real-world workloads. For instance, *mail* has only 7.83% unique write requests, representing a huge potential for de-duplicating the remaining 2.63 million writes. Similarly, *web* and *homes* can provide 57.65% and 33.63% write reductions respectively, improving the performance and lifetime of SSDs substantially. Furthermore, only a small fraction of writes in these workloads are due to same values being written at the same locations. For example, about 8% overall writes in *mail* and *homes* are due to same LPN writing the same content successively. A majority of duplicate writes are attributed to same content being written to different locations requiring sophisticated CAS-based scheme for de-duplication. In Figure 2, we present VP (as CDFs) for reads and writes for the three workloads. A given point on the x-axis can correspond to different values for reads/writes.

The following insights and observations emerge from our definition and these statistics. *First*, all these workloads exhibit significant skewness in VP, i.e, a small fraction of total values account for large number of accesses. For example, the fraction of total unique values that account for 50% of the overall writes are 14.44%, 8.84%, and 29.99% for *homes*, *mail* and *web* respectively(shown by dotted lines). Therefore, pinning these values in the meta-data cache can offer write traffic reduction of 35.56%, 41.16%, and 20.01%, respectively. Similar benefits apply for reads upon caching the most popular (value, content) pairs in the content cache. *Second*, we find that these workloads exhibit different degrees of value popularity (e.g., *homes* has higher VP for reads than *mail*, while *mail* has higher VP for writes) implying different degrees of potential benefits for reads/writes.

Temporal Value Locality (TVL): The presence of TVL in a workload implies that if a certain value (as opposed to LPN) is accessed now, it is likely to be accessed again in the near future, not necessarily by the same LPN. We distinguish TVL for writes and reads to be able to differentiate benefits that could be obtained



(a) web (b) mail (c) homes

Figure 3: Temporal value locality and temporal locality(labeled LPN) for writes in real-world workloads (1 day traces). We show the meta-data cache size that can contribute to 90% of the total writes.

from the use of meta-data vs. content caches. We modify a standard way of characterizing LPN-based temporal locality for representing TVL [21]. For each workload, assuming the meta-data cache to be managed as a queue with a least-recently-used (LRU) eviction policy for values, we present CDFs of number of writes of the value at the $(i+1)^{st}(i \geq 0)$ location within the LRU queue in Figure 3.

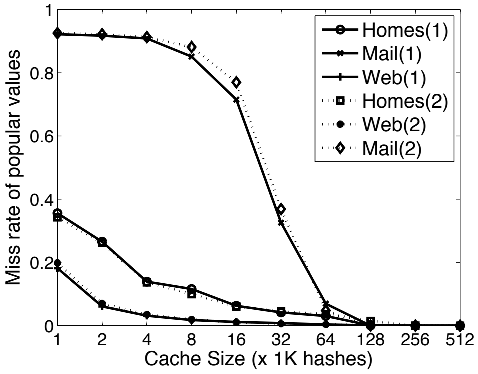


Figure 4: Cache miss rate for popular values. The number in brackets represent the length of the trace in number of days. Note that popular values denotes the minimum number of values which account for 50% of accesses in the workload. The cache size on the X-axis (logscale) is in terms of 1K hashes.

Implications for writes: The presence of TVL for writes implies that even a small meta-data cache could achieve high hit rates to provide write reduction. For example, the maximum size of the meta-data cache required for storing all the values in the 1 day trace of *homes* is around 7.5MB (each entry in this cache requires 28B for storing the hashing structures as we explain in Section 4.1). However, 90% of writes for *homes* are satisfied within 11046 positions in the LRU queue requiring only about 600KB in the meta-data cache, thus reducing the space requirements by about 96%. Even *mail* which shows lesser TVL provides savings of approximately 65% for achieving 90% hit rate.

Clearly the size of meta-data cache affects these gains. Figure 4 shows the miss rate for popular value lookups done for writes as a function of different sizes of this cache. Additionally, we use portions of the workloads over 1-day and 2-day periods and find that TVL sustains over this duration. We find that for our workloads, a LRU cache based on TVL is able to hold popular values, thus offering an easy way to implement a technique that can recognize VP. Whereas in our workloads, TVL and skewness in VP occur together, generally speaking, these could be mutually exclusive. For example, it may be the case that for a workload with high TVL, all values are equally popular, i.e., have comparable number of write accesses, thus displaying low skewness in VP. Alternately, a workload with high skewness(in VP) can exhibit low TVL if the popular values have a long time gap between successive accesses. We design CA-SSD so that it can exploit these properties whenever present, but not experience degraded performance (compared to a regular SSD) when these are absent.

Implications for reads: We observe higher TVL than temporal locality even for reads suggesting that, for these workloads, a value-based content cache is likely to outperform one using LPNs and offer reduction in read traffic to the SSD. Similar observation was made in [26] for developing a content based cache for improving I/O performance in the context of HDD-based storage.

Finally, one could also consider a notion of *spatial value locality* (SVL). The principal of spatial locality, as used conventionally, can be stated as follows: if (content corresponding to) a logical address X is accessed now, addresses in the neighborhood of X are likely to be accessed in the near future. SVL emerges from a generalized take on what the neighborhood or proximity of a data item means. It posits that a given value, even when part of different logical data items, is likely to see similarities among the values in its neighborhood. Stated another way, spatial value locality hypothesizes that there might exist positive correlations among certain values in terms of their closeness with respect to their addresses within (possibly multiple) logical data objects. SVL has been used for handling disk bottleneck for meta-data management in CAS systems for backup applications by prefetching key-value pairs which are accessed together [48]. For SSDs, it can provide additional benefits for reads when sub-page level chunks are used. We do not explore SVL or other optimizations for reads in this work and leave it as part of our future work.

4 Design of CA-FTL

We develop the CA-FTL mapping unit and GC based on the issues discussed in Section 2. We assume a CAS chunk unit to be equal to the flash page size.

4.1 The CA-FTL Mapping Unit

Address Translation and Meta-data Management: As discussed in Section 2, CA-FTL requires additional data structures for maintaining information about hashes and their relationship with LPNs. Figure 5(b) shows the data structures we employ to realize CA-FTL's Mapping Unit. We assume address translations to be kept at the granularity of a page. Such page-level mappings have been shown to be desirable and scalable in recent research [16, 25]. *First*, similar to existing FTLs, we have a table called LPT which stores translations between LPNs to PPNs. Each entry requires 4B for storing the LPN and another 4B for PPN. Thus, the maximum space needed for LPT in a 4GB SSD is 8MB (for 100% flash utilization). *Second*, an inverted LPT (iLPT) stores the list of LPNs that correspond to the same value and thus the same PPN. The iLPT is used to keep track of valid values. If the LPN list for a PPN is empty, it signifies that no LPN stores the value present in that PPN and the page should be invalidated. The iLPT is queried during GC and WL for updating the LPT whenever the PPN storing a value changes. *Third*, we use a hash-to-PPN table (HPT) to store hash to PPN mappings that is looked up on a write request to decide whether the write is for an existing value (no flash write needed) or a new value (requires a flash write). Entries are inserted or updated in the HPT upon (i) a write request with a new value or (ii) a page write due to GC/WL, respectively. Page invalidations result in removal of entries. Each hash is 16-20B long depending on the hashing algorithm used (16B for MD5 and 20B for SHA1) whereas a PPN is 4B long. For a 4GB SSD, the maximum space needed for the HPT is 20-24MB since the maximum number of PPNs it can store is 1M. All further discussion is in context with MD5 hashes present in the available real-world workloads but our ideas apply readily for SHA1 hashes also. *Fourth*, we employ an inverted HPT (iHPT) which maps PPNs to hashes by storing the addresses of the corresponding HPT entries. It stores the same number of valid entries as HPT. When a flash page is invalidated, iHPT provides the address of the corresponding HPT entry to be removed without incurring an OOB read.

Let us now understand how to deal with space overheads of these data structures. (i) Gupta et al. [16] have proposed page based FTL which exploits temporal locality in workloads to reduce the LPT space requirements. As shown in Figure 3, real-world workloads also demonstrate significant temporal locality apart from TVL. Thus, we can utilize variants of page-based FTLs to reduce the space requirements by only storing a subset of the LPT/iLPT in our BB-RAM. (ii) Since the HPT/iHPT's space needs can be prohibitively large (recall that on a 4GB flash, they require up to 28MB of RAM), we are forced to only store a subset. Given our findings about the presence of TVL in workloads, we implement the HPT as a cache of hash-to-PPN mappings employing a LRU eviction policy for writes of values. The size of this cache could be chosen by CA-FTL based on how much RAM it can afford to use for meta-data storage. When all of this cache is occupied, to insert a new entry we discard the least-recently-used entry from the HPT and the iHPT. A salient aspect of our strategy is that, unlike a traditional LRU-based queue, we do not maintain the remainder of the HPT/iHPT (which does not fit in RAM) on another storage medium (e.g., the flash medium itself). On an entry's eviction from the meta-data cache, we simply discard it. This saves us potential flash page writes (write-back of evicted dirty entries) and reads (mapping entry lookup on a HPT/iHPT miss). This scheme trades off reduction in RAM occupied for meta-data for a reduction in the degree of data de-duplication achieved, since some values may be re-written upon HPT misses. Our findings on TVL in real workloads in Section 3 suggest that such misses are likely to be rare even for nominal cache sizes. This is shown in Figure 4 where a cache size of 1.75MB (for storing 64K hashes) yields miss rates less than 7% for *mail*. For *web* and *homes*, these miss rates are even smaller, being 0.4% and 4%, respectively. Thus, most of the discarded entries correspond to less popular values which have low write frequency and less impact on de-duplication efficiency. In Section 5, we evaluate the performance of CA-SSD with different meta-data cache sizes. Data/meta-data consistency is not impacted due to this scheme since the LPT which stores the LPN-to-PPN mappings required for managing consistency (explained earlier in Section 2.2) is managed independent of this strategy. Furthermore, BB-RAM is only needed for persistent storage of

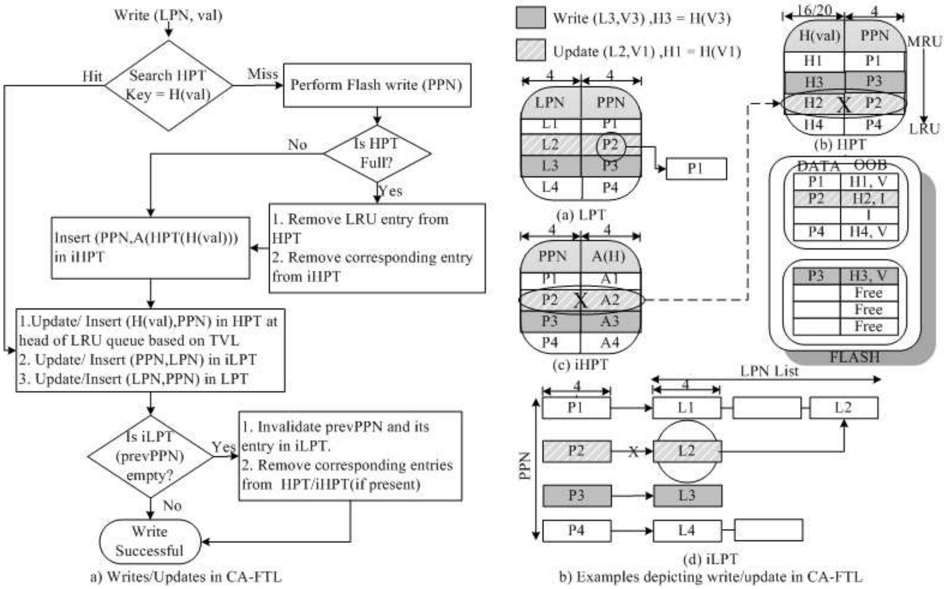

Write (LPN, val)
Search HPT Key = H(val)
Hit
Miss
Perform Flash write (PPN)
Is HPT Full?
No
Yes
1. Remove LRU entry from HPT
2. Remove corresponding entry from iHPT
Insert (PPN,A(HPT(H(val))) in iHPT
1.Update/ Insert (H(val),PPN) in HPT at head of LRU queue based on TVL
2. Update/ Insert (PPN,LPN) in iLPT
3. Update/Insert (LPN,PPN) in LPT
Is iLPT (prevPPN) empty?
Yes
No
1. Invalidate prevPPN and its entry in iLPT.
2. Remove corresponding entries from HPT/iHPT(if present)
Write Successful
a) Writes/Updates in CA-FTL
Write (L3,V3) ,H3 = H(V3)
Update (L2,V1) ,H1 = H(V1)
LPN
PPN
L1 P1
L2 P2
L3 P3
L4 P4
P1
(a) LPT
PPN
A(H)
P1 A1
P2 A2
P3 A3
P4 A4
(c) iHPT
16/20
4
H(val)
PPN
H1 P1
H3 P3
H2 P2
H4 P4
MRU
LRU
(b) HPT
DATA OOB
P1 H1, V
P2 H2, I
I
P4 H4, V
P3 H3, V
Free
Free
Free
FLASH
LPN List
PPN
P1 L1 L2
P2 L2
P3 L3
P4 L4
(d) iLPT
b) Examples depicting write/update in CA-FTL

Figure 5: (a) Flowchart depicting how writes are handled by CA-FTL. *val* represents the content to be written. (b) Example of write requests: (1) Write request (L3,V3) to a new LPN L3 with a new value V3 results in a flash page write (P3). Entries are added to all four data structures. (2) Update (L2,V1) results in a HPT hit for H1, the entry is moved to the head of LRU queue(based on TVL) in HPT. L2 is then added to the the LPN list for P1 in the iLPT and removed from P2's list. Since P2's list (in the iLPT) is now empty, the flash page (P2) is invalidated and the corresponding entries in HPT and iHPT are removed. (Note that iHPT only stores the address of the corresponding HPT entry and not the complete hash.)

LPT whereas other mapping structures can be stored on volatile RAM without impacting consistency.

Handling Read/Write Requests: Read requests in CA-FTL are handled similar to traditional FTLs. LPT is looked up to locate the PPN storing the value and its contents are returned to the upper layers. The flowchart in Figure 5(a) describes the handling the write/update requests in CA-FTL. On receiving a write request, the hash of the value for each LPN comprising the request is calculated and the HPT is then looked up with this hash. A miss is deemed to indicate request for a new value and a flash page write is issued. If the HPT is fully occupied, the least recently used entry is discarded and the new (hash, PPN) entry is inserted at the head of the LRU queue based on TVL. Corresponding updates are made in the iHPT also. Finally, the LPT and the iLPT are updated. On a hit in the HPT, the entry is moved to the head of the LRU queue and LPT/iLPT are updated. Furthermore, update requests may result in LPN storing a different value, requiring modifications to the mapping entries for the LPN's earlier value. If the LPN list in iLPT for the PPN corresponding to the LPN's earlier value is empty, the entry and the physical page on flash are invalidated. Finally, the HPT/iHPT entries for this PPN are also removed (Note that the eviction strategy may have already discarded the HPT/iHPT entries, hence not requiring an explicit removal). Figure 5(b) gives examples describing the handling of writes in CA-FTL including relevant meta-data cache management.

4.2 Garbage Collection in CA-FTL

Unlike conventional SSDs where all writes are propagated to flash, CA-SSD only requires one write per unique value (W_{unique}) except in the case of meta-data cache misses (due to limited cache size) where some duplicate values (W$_{dup}$) may also be written. Writes may also be needed for values which have been invalidated/erased (when no LPN points to them) and are reborn (W_{reborn}) due to subsequent write requests. Similar to conventional SSDs, the final component is GC writes (W_{gc}) which depends on the number of GC invocations as well as the number of valid pages copied upon each such invocation. Therefore, the total writes for a CA-SSD can be expressed as a sum of these components: $W_{total} = W_{unique} + W_{dup} + W_{reborn} + W_{gc}$.

In traditional SSDs, every LPN update results in invalidation of the PPN containing the previous LPN version. CA-SSD only invalidates pages when the value in them becomes *dead* in the sense of no LPN being associated with it any longer. Thus, garbage is likely to be gener-

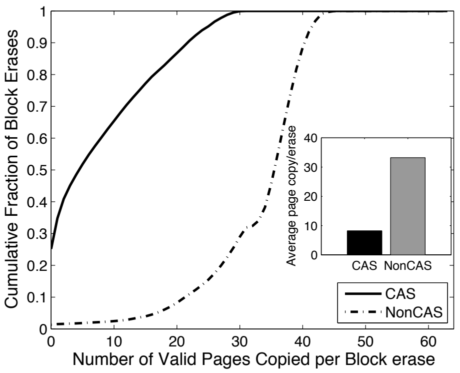


Figure 6: Cumulative distribution of valid pages in blocks erased during GC in *web* workload.

ated at a slower rate in CA-SSD. This coupled with the reduction in write traffic to flash due to de-duplication decreases the number of GC invocations for the same GC policy as in a traditional SSD. The other aspect is the number of pages copied during GC. As shown in Figure 6 for *web*, the valid content in the victim blocks is much lower in CA-SSD as compared to that in traditional SSD. The average number of pages copied per block decreases from 33.20 to 8.21, a reduction of about 75.27% with CA-SSD. This is primarily due to data de-duplication which reduces the amount of total valid content stored on flash, in turn increasing the fraction of invalid pages in victims. These observations lead us to conclude that existing GC mechanisms should work well even in a CA-SSD. We evaluate the impact of our choice in Section 5.

5 Experimental Evaluation

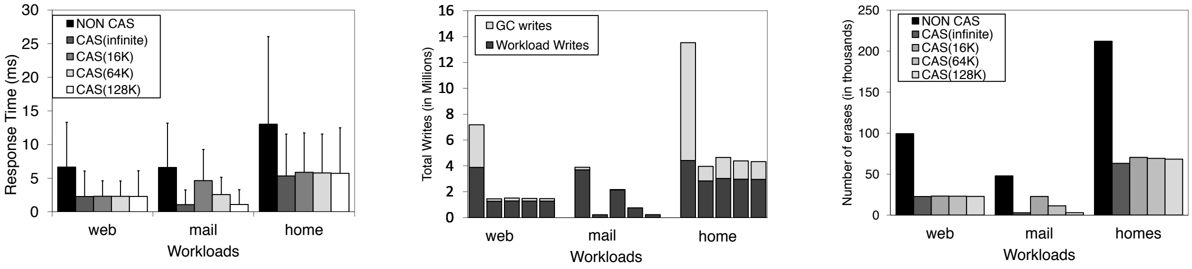
5.1 Experimental Setup

We simulate both traditional and CA-SSDs using SSD simulator [10] which has been integrated into Disksim-4.0 [19]. The SSD simulator is capable of simulating both SLC and MLC SSDs with multiple planes and dies. As described in Section 2, we use SLC SSDs with extra large pages(SLC2) and single plane in this study (refer to Table 1 for SSD properties). We have modified the Disksim interface to use block-based traces with content hashes. We have implemented the FTL for our CA-SSD (CA-FTL) with the meta-data cache manintained using LRU eviction based on TVL. We simulate the hashing unit in CA-SSD by modeling the overheads (32μs [18]) of performing hash calculation along with their impact on the queueing delays at the SSD controller. Note that this is a conservative estimate and the hash calculation overheads are likely to be much lower in CA-SSD (As discussed in Section 2.2, SSDs with crypto-units have reported similar performance to traditional SSDs [5]). As explained earlier, we do not simulate read caching in either traditional or CA-SSD.

5.2 Real-world Traces

We first focus on the three real workload traces that were found to exhibit high VL in Section 3. Figure 7(a) shows the mean response time comparing the standard SSD with two CA-SSD configurations: (i) sufficient capacity in its RAM to store HPT/iHPT and (ii) capacity to store only a fixed number of hashes in RAM. For example, storing 128K hashes in HPT/iHPT requires 3.5MB. We also present mean response times for other meta-data cache configurations. We note the tremendous performance benefits obtained with our CA-SSD compared to the traditional SSD and the benefits directly correlate to the value locality/popularity in writes. For instance, the *mail* workload, which in Figure 2(b) demonstrates the highest VP of the three for writes, shows a 84% reduction in response time with CA-SSD compared to the traditional SSD. The reductions are substantial for *homes* and *web* as well, which show 59% and 65% improvements in response times.

In order to understand these benefits further, we break down the write traffic into those that are (a) directly imposed by the workload and (b) additional writes imposed due to GC when valid pages need to be copied across blocks. The number of writes in each category is shown in Figure 7(b) for the traditional SSD and our CA-SSD. Overall, the reductions in write traffic for CA-SSD are 77%, 93% and 70% for *web*,*mail* and *homes*, respectively over a traditional SSD. We see significant reductions in writes of both categories. The drop in category (a) is intuititve to follow given the value popularity in the workloads. Additionally, there is significant reduction in category (b) writes as well - 94%, 100% and 87% for *web*, *mail* , and *homes*, respectively. In fact, in percentage terms, these GC write reductions overshadow the category (a) reductions. Note that GC overhead is a function of the amount of garbage in the flash, and the distribution of this garbage across the blocks. Since a page in CA-SSD is treated as invalid only when all the LPNs having that content have written a “different value,” it is less likely to be marked as garbage compared to a traditional SSD where “any” (including the prior identical) LPN write necessitates a page invalidation. Furthermore, the decrease in the amount of valid content on the SSD due to de-duplication directly reduces pages copied during GC. All these reasons contribute to the substantial benefits that CA-SSD experiences in lower induced writes/copies compared to a traditional SSD. In fact, for the *mail* workload we observe no GC writes since the total number of unique values seen for this workload fits



(a) Response Time

(b) Total Writes

(c) Block Erases

Figure 7: Performance of CA-SSD vs traditional SSDs. (b)The reborn writes fraction is extremely low and hence not shown. The bars for each workload should be read in the following order: NonCAS, CAS(infinite), CAS(16K), CAS(128K), CAS(256K). Note that CAS(x) represents the meta-data cache size in terms of number of hashes(x) it can store. For response times, we also present the standard deviation, and observe that CA-SSD offers reduction in the variance in addition to the average.

within the chosen SSD size without triggering GC.

Another important characteristic is the lifetime of SSD which depends on the write-erase cycles of blocks. Higher incoming write traffic results in higher block erases, reducing the useful lifetime of SSD. Write reduction benefits from CA-SSD on both workload and GC writes directly translate into reduced block erases. As shown in Figure 7(c), the number of block erases in *mail* reduces from 47819 to 2876, more than 15-fold decrease. Similarly, *homes* and *web* experience 70% and 77% reductions in block erases, respectively.

In Figure 7, we showed results for CA-SSD with both unlimited RAM capacity to store the HPT/iHPT, as well as finite capacities of 16K, 128K, and 256K entries that require about 450KB, 3.5MB and 7MB of space respectively. Even for meta-data cache capacities less than 1MB, CA-SSD shows significant improvements over traditional SSD. For example, the mean response time for *homes* decreases by about 7ms (for 16K hashes) in CA-SSD as compared to traditional SSD whereas the block erases reduce by 65%. As we had seen in Section 3, *mail* shows lower TVL for writes and hence requires a larger meta-data cache to exploit CA-SSD benefits. However, we note that beyond 128K entries, we observe close to the infinite CA-SSD behavior for all workloads, reiterating the observations made in Section 3 regarding the ability to hold a substantial portion of the working set of the meta-data in these workloads within a relatively small space because of presence of TVL. 3.5MB of RAM is a relatively small amount of space to support in today's SSDs - for instance, a 1TB [6] SSD has 512MB of DRAM which can be used for storing the meta-data. Regardless of the actual amount of available space to store this meta-data, CA-SSD can avail of whatever space is allocated to it, and as we will show in the next subsection, even "complete absence of value locality" makes CA-SSD only slightly worse than a traditional SSD.

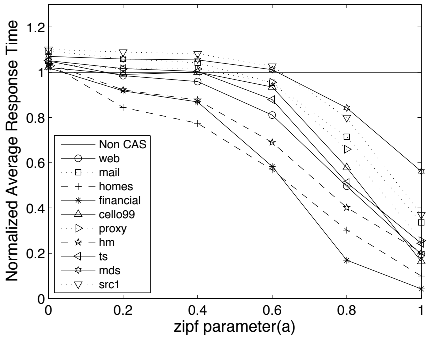


Figure 8: Impact of VL. zipf parameter on X-axis represents the extent of VP skewness in the workload. Higher zipf paramter indicates higher skewness in VP. The average response times on Y-axis are normalized with respect to average response times for traditional SSDs. Note that these response times are for unlimited cache-space. We observe similar response times for meta-data cache which can store 128K hashes.

5.3 Impact of Value Locality on CA-SSD

We next conduct a more extensive analysis of the impact of value locality on CA-SSD performance to demonstrate that it is beneficial across a broad spectrum of workload behaviors and not just for the three real workloads used above which exhibit good value locality. One difficulty in considering a wide range of workloads is the lack of real workload traces for which content of each write is made available in the trace (most traces contain just the timestamp, address and size fields). On the other hand, considering a purely synthetic workload,

Workload	Description	Size (GB)	Requests (in mill.)	% Writes
financial	OLTP	0.50	6.50	79.60
cello99	HP-UX OS	0.46	0.44	70.79
proxy	Proxy server	0.33	2.44	95.64
hm	H/W Monitor	2.43	11.11	54.74
ts	Terminal Server	0.91	4.17	74.06
mds	Media Server	3.09	2.89	70.46
src1	Source Control	1.47	5.00	93.73

Table 3: Workload description. Apart from the above 7 workloads, we use *mail*, *web*, and *homes* that were described in Section 3. The workload size represents the total number of unique logical addresses(LPNs) accessed in the trace. The logical address space exposed to the file system can be much larger.

may mandate assumptions on parameters - such as arrival rate, sequentiality, temporal locality, etc. - over and beyond those pertaining to value locality. Instead, we pick a set of 10 real workload traces(refer to Table 2 and Table 3) that have been studied in prior literature - *financial* from UMass [8], *cello99* from HP Labs [2], *proxy,hm,ts,mds* and *src1* from MSR [9] including the three workloads(*homes,web* and *mail*) from FIU [26] . We use the arrival times, block addresses and sizes from these traces, and only synthesize the "content"(v) for the blocks using a *zipf* distribution, given as: $P(v_i) = Cv_i{}^{-a}$, where, $C = 1/\sum_{i=1}^{N} v_i^{-a}$, N is the total unique values in the workload and a is the zipf parameter representing the skewness in value popularity. Many prior studies [13] have shown content popularity can be characterized by this distribution. Furthermore, we vary the exponent(a) characterizing the distribution from 0 (which corresponds to no VP) to 1.0 (which corresponds to a very highly skewed VP behavior). In the experiments, we use this zipf probability distribution to pick a value for each incoming request. This exercises only the popularity of values and ignores the spatial and temporal dimensions of value locality, and can thus be viewed as a pessimistic evaluation of CA-SSD since any spatial/temporal VL will only benefit it further (and not affect the performance of a traditional SSD which only relies on LPN-based spatial/temporal locality). Figure 8 shows mean response times for these workloads on CA-SSD normalized with respect to traditional SSD response times. Similar to results in Section 5.2, as VP increases, the response times for these workloads decreases. Furthermore, even when VP is low, the response times for CA-SSD and traditional SSDs are comparable. We observe that when the workloads show no VP (a=0.0), the average response time of CA-SSD only increases by atmost 10% (for src1). This is primarily due to the overheads of the hashing unit for write requests which we have chosen conservatively. Thus, we expect the average response time to be lower with a more aggressive estimate (If needed, one could even explore the possibility of dynamically turning off CAS in CA-SSD in complete absence of VL). On the other hand, for high VP ($a = 1.0$), we see tremendous benefits with CA-SSD. We observe around 25 times reduction in average response times for financial trace and on average all workloads show an improvement of about 74%. Furthermore, the number of values which account for 50% of the write requests in hm workload decreases from 4.5M for no VP ($a = 0.0$) to 1.3M for moderate VP ($a = 0.4$), a reduction of approximately 71%. This clearly illustrates that the benefits accrued through VP specifically and value locality in general, strengthen the case for adoption of content addressability in SSDs, paving the way for a new generation of SSDs.

6 Related Work

Value Locality/Content Addressability: CAS has been extensively used in archival and backup systems such as Venti [42], Foundation [43], Pastiche [14] etc for space savings , Internet suspend/resume [27], LBFS [35] for saving network bandwidth file system and buffer cache design [35, 47, 34], etc. Some recent work has evaluated real-world workloads and demonstrated significant value locality which bodes well for CA-SSD [26, 36]. However, to the best of our knowledge, this paper is the first to focus on issues that arise when designing an SSD that uses CAS internally.

Meta-data Management for CAS: The scalability of a system employing CAS depends on careful management of CAS related meta-data. Larger-sized chunks help in reducing the amount of meta-data to be stored while smaller chunks provide good duplicate elimination. Pasta [33], Pastiche [14], REBL [29] and Foundation [43] have explored more complex chunking methods. Bimodal chunking attempts to combine the benefits of two different chunk sizes [28] CA-SSD could benefit from all of these techniques and evaluating the benefits of different/variable chunk sizes is part of our future work. Sparse indexing divides the incoming data stream into large segments which are then de-duplicated against a few similar segments found using sampling [31]. Like sparse indexing, the degree of de-duplication in CA-SSD depends on the available meta-data cache space. Researchers have developed CAS meta-data management techniques which utilize HDD/SSDs for storing chunk indexes [48, 15]. These techniques utilize spatial locality in data segments for reducing index lookups by pre-fetching meta-data in RAM. Unlike these techniques, CA-SSD does away with index lookups on HDD/SSD and utilizes TVL for reducing meta-data misses.

7 Conclusion

Given evidence for the presence of significant VL in real-world workloads, we designed CA-SSD which employed CAS for its internal data management. Using three real-world workloads with content information, we devised statistical characterizations of two aspects of VL - value popularity and temporal VL - that formed the foundation of CA-SSD. The design of CA-SSD presented us with interesting choices and challenges related to exploiting VL for write reduction and maintaining meta-data consistency under constrained cache space. Using several real-world workloads, we conducted an extensive evaluation of CA-SSD. We found significant improvements (59-84%) in average response times. Even for workloads with little or no value locality, we observed that CA-SSD continued to offer comparable performance to a traditional SSD.

Acknowledgments

We would like to express our gratitude to our shepherd Ohad Rodeh of IBM Almaden Research Center and the anonymous reviewers for their detailed comments that helped us improve the quality of our paper. This research was supported in part by NSF grants CCF-0811670, CNS-0720456, CNS-0615097, and CAREER award CNS-0953541.

References

[1] 64MB Cache on SSD. http://www.tomshardware.com/news/A-DATA-OCZ-64MB-Cache,7263.html.

[2] HP Labs. Tools and Traces. http://tesla.hpl.hp.com/public_software.

[3] HP Memory Smart Array Controller. http://www1.hp.com.

[4] Intel's 3rd Generation X25-M SSD Specs Revealed. http://www.anandtech.com/show/3965/intels-3rd-generation-x25m-ssd-specs%-revealed.

[5] OCZ Vertex 2 EX Series SATA II 2.5" SSD. http://www.ocztechnology.com.

[6] OCZ Z-Drive R2 e88 PCI-Express SSD. http://www.ocztechnology.com.

[7] Raw Drive Capacity Cost Trends. http://wikibon.org/w/images/a/a4/EMCRawDriveCapacityCostTrends.jpg.

[8] UMass Trace Repository, 2007. http://traces.cs.umass.edu.

[9] SNIA. IOTTA repository, January 2009. http://iotta.snia.org/.

[10] AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J. D., MANASSE, M. S., AND PANIGRAHY, R. Design Tradeoffs for SSD Performance. In *ATC 08: Proceedings of the USENIX Annual Technical Conference* (2008).

[11] ARPACI-DUSSEAU, A., ARPACI-DUSSEAU, R. H., AND PRABHAKARAN, V. Removing The Costs Of Indirection in Flash-based SSDs with Nameless Writes. In *HotStorage 10: Proceedings of the 2nd Workshop on Hot Topics in Storage and File Systems* (2010).

[12] BLACK, J. Compare-by-hash: a reasoned analysis. In *ATC '06: Proceedings of the USENIX '06 Annual Technical Conference* (2006).

[13] CHERVENAK, A. L. Challenges for tertiary storage in multimedia servers. *Parallel Computing* (1998).

[14] COX, L. P., MURRAY, C. D., AND NOBLE, B. D. Pastiche: Making Backup Cheap and Easy. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation* (2002).

[15] DEBNATH, B., SENGUPTA, S., AND LI, J. ChunkStash: Speeding up Inline Storage Deduplication using Flash Memory. In *ATC'10: Proceedings of the USENIX 2010 Annual Technical Conference* (2010).

[16] GUPTA, A., KIM, Y., AND URGAONKAR, B. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems* (2009).

[17] HANDY, J. PCM becomes a reality, 2009. http://www.objective-analysis.com.

[18] HELION. Fast hashing cores. http://www.heliontech.com/fast_hash.htm.

[19] JOHN S. BUCY, J. S., SCHLOSSER, S. W., AND GANGER, G. R. *The DiskSim Simulation Environment Version 4.0 Reference Manual.* http://www.pdl.cmu.edu/DiskSim/.

[20] JUNG, D., CHAE, Y.-H., JO, H., KIM, J.-S., AND LEE, J. A group-based wear-leveling algorithm for large-capacity flash memory storage systems. In *CASES '07: Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded systems* (2007).

[21] KAREDLA, R., LOVE, J. S., AND WHERRY, B. G. Caching strategies to improve disk system performance. *Computer 27*, 3 (1994), 38–46.

[22] KAWAGUCHI, A., NISHIOKA, S., AND MOTODA, H. A flash-memory based file system. In *TCON'95: Proceedings of the USENIX 1995 Technical Conference* (1995).

[23] KGIL, T., AND MUDGE, T. N. FlashCache: a NAND flash memory file cache for low power web servers. In *CASES 06: Proceedings of the 2006 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems* (2006).

[24] KIM, H., AND AHN, S. BPLRU: A buffer management scheme for improving random writes in flash storage. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies* (2008).

[25] KIM, J. K., LEE, H. G., CHOI, S., AND BAHNG, K. I. A PRAM and NAND flash hybrid architecture for high-performance embedded storage subsystems. In *EMSOFT 2008: Proceedings of the 8th ACM & IEEE International conference on Embedded software* (2008).

[26] KOLLER, R., AND RANGASWAMI, R. I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance. In *FAST'10: Proceedings of the 8th USENIX Conference on File and Storage Technologies* (2010).

[27] KOZUCH, M., AND SATYANARAYANAN, M. Internet suspend/resume. In *WMCSA '02: Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications* (2002).

[28] KRUUS, E., UNGUREANU, C., AND DUBNICKI, C. Bimodal Content Defined Chunking for Backup Streams. In *FAST'10: Proceedings of the 8th USENIX Conference on File and Storage Technologies* (2010).

[29] KULKARNI, P., DOUGLIS, F., LAVOIE, J., AND TRACEY, J. M. Redundancy elimination within large collections of files. In *ATC '04: Proceedings of the USENIX Annual Technical Conference* (2004).

[30] LEE, S.-W., AND MOON, B. Design of flash-based DBMS: an in-page logging approach. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data* (2007).

[31] LILLIBRIDGE, M., ESHGHI, K., BHAGWAT, D., DEOLALIKAR, V., TREZISE, G., AND CAMBLE, P. Sparse indexing: large scale, inline deduplication using sampling and locality. In *FAST '09: Proccedings of the 7th USENIX conference on File and storage technologies* (2009).

[32] LOFGREN, K. M. J., NORMAN, R. D., THELIN, G. B., AND GUPTA, A. Wear leveling techniques for flash EEPROM systems. In *United States Patent, No 6850443* (2005).

[33] MORETON, T. D., PRATT, I. A., AND HARRIS, T. L. Storage, Mutability and Naming in Pasta. In *Revised Papers from the NETWORKING 2002 Workshops on Web Engineering and Peer-to-Peer Computing* (2002).

[34] MORREY, C. B., AND GRUNWALD, D. Content-Based Block Caching. In *MSST 06: 23rd IEEE, 14th NASA Goddard Conference on Mass Storage Systems and Technologies* (2006).

[35] MUTHITACHAROEN, A., CHEN, B., AND MAZIÈRES, D. A low-bandwidth network file system. In *SOSP '01: Proceedings of the 18th ACM Symposium on Operating systems principles* (2001).

[36] NATH, P., URGAONKAR, B., AND SIVASUBRAMANIAM, A. Evaluating the Usefulness of Content-Addressable Storage for High-Performance Data-Intensive Applications. In *HPDC 08: Proceedings of the ACM/IEEE International Symposium on High Performance Distributed Computing* (Jun 2008).

[37] NUMONYX MEMORY SOLUTIONS. *16-Gbit MLC NAND flash memories.* http://numonyx.com/Documents/Datasheets/NAND16GW3D2B.pdf.

[38] NUMONYX MEMORY SOLUTIONS. *2-Gbit SLC NAND flash memories.* http://numonyx.com/Documents/Datasheets/NAND02G-BxD.pdf.

[39] NUMONYX MEMORY SOLUTIONS. *64-Gbit SLC NAND flash memories.* http://numonyx.com/Documents/Datasheets/NAND64GW3FGA.pdf.

[40] PRIMMER, R., AND HALLUIN, C. D. Collision and preimage resistance of the centera content address. Tech. rep., 2005.

[41] PURESILICON. Puresi 1TB SSD with hardware based encryption. http://www.marketwire.com.

[42] QUINLAN, S., AND DORWARD, S. Venti: A new approach to archival data storage. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies* (2002).

[43] RHEA, S., COX, R., AND PESTEREV, A. Fast, inexpensive content-addressed storage in foundation. In *ATC'08: Proceedings of the USENIX 2008 Annual Technical Conference* (2008).

[44] SAMSUNG. Samsung self encrypting ssd. "http://www.engadget.com/2009/04/16/samsung-comes-clean-with-self-encry%pting-ssds.

[45] SMULLEN, C. W., COFFMAN, J., AND GURUMURTHI, S. Accelerating enterprise solid-state disks with non-volatile merge caching. In *IGCC'10: Proceedings of the 1st International Conference on Green Computin* (2010).

[46] SOUNDARARAJAN, G., PRABHAKARAN, V., BALAKRISHNAN, M., AND WOBBER, T. Extending SSD Lifetimes with Disk-Based Write Caches. In *FAST 10: Proceedings of the 8th USENIX Conference on File and Storage Technologies, 2010* (2010).

[47] VILAYANNUR, M., NATH, P., AND SIVASUBRAMANIAM, A. Providing Tunable Consistency For a Parallel File Store. In *FAST05: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies* (2005).

[48] 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* (2008).

Reliably Erasing Data From Flash-Based Solid State Drives

Michael Wei[*], *Laura M. Grupp*[*], *Frederick E. Spada*[†], *Steven Swanson*[*]

[*]*Department of Computer Science and Engineering, University of California, San Diego*

[†]*Center for Magnetic Recording and Research, University of California, San Diego*

Abstract

Reliably erasing data from storage media (*sanitizing* the media) is a critical component of secure data management. While sanitizing entire disks and individual files is well-understood for hard drives, flash-based solid state disks have a very different internal architecture, so it is unclear whether hard drive techniques will work for SSDs as well.

We empirically evaluate the effectiveness of hard drive-oriented techniques and of the SSDs' built-in sanitization commands by extracting raw data from the SSD's flash chips after applying these techniques and commands. Our results lead to three conclusions: First, built-in commands are effective, but manufacturers sometimes implement them incorrectly. Second, overwriting the entire visible address space of an SSD twice is usually, but not always, sufficient to sanitize the drive. Third, none of the existing hard drive-oriented techniques for individual file sanitization are effective on SSDs.

This third conclusion leads us to develop flash translation layer extensions that exploit the details of flash memory's behavior to efficiently support file sanitization. Overall, we find that reliable SSD sanitization requires built-in, verifiable sanitize operations.

1 Introduction

As users, corporations, and government agencies store more data in digital media, managing that data and access to it becomes increasingly important. Reliably removing data from persistent storage is an essential aspect of this management process, and several techniques that reliably delete data from hard disks are available as built-in ATA or SCSI commands, software tools, and government standards.

These techniques provide effective means of sanitizing hard disk drives (HDDs) – either individual files they store or the drive in their entirety. Software methods typically involve overwriting all or part of the drive multiple times with patterns specifically designed to obscure any remnant data. The ATA and SCSI command sets include "secure erase" commands that should sanitize an entire disk. Physical destruction and degaussing are also effective.

Flash-based solid-state drives (SSDs) differ from hard drives in both the technology they use to store data (flash chips vs. magnetic disks) and the algorithms they use to manage and access that data. SSDs maintain a layer of indirection between the logical block addresses that computer systems use to access data and the raw flash addresses that identify physical storage. The layer of indirection enhances SSD performance and reliability by hiding flash memory's idiosyncratic interface and managing its limited lifetime, but it can also produce copies of the data that are invisible to the user but that a sophisticated attacker can recover.

The differences between SSDs and hard drives make it uncertain whether techniques and commands developed for hard drives willl be effective on SSDs. We have developed a procedure to determine whether a sanitization procedure is effective on an SSDs: We write a structured data pattern to the drive, apply the sanitization technique, dismantle the drive, and extract the raw data directly from the flash chips using a custom flash testing system.

We tested ATA commands for sanitizing an entire SSD, software techniques to do the same, and software techniques for sanitizing individual files. We find that while most implementations of the ATA commands are correct, others contain serious bugs that can, in some cases, result in all the data remaining intact on the drive. Our data shows software-based full-disk techniques are usually, but not always, effective, and we have found evidence that the data pattern used may impact the effectiveness of overwriting. Single-file sanitization techniques, however, consistently fail to remove data from the SSD.

Enabling single-file sanitization requires changes to the flash translation layer that manages the mapping between logical and physical addresses. We have developed three mechanisms to support single-file sanitization and implemented them in a simulated SSD. The mechanisms rely on a detailed understanding of flash memory's behavior beyond what datasheets typically supply. The techniques can either sacrifice a small amount of performance for continuous sanitization or they can preserve common case performance and support sanitization on demand.

We conclude that the complexity of SSDs relative to hard drives requires that they provide built-in sanitization commands. Our tests show that since manufacturers do not always implement these commands correctly, the commands should be verifiable as well. Current and proposed ATA and SCSI standards provide no mechanism for verification and the current trend toward encrypting SSDs makes verification even harder.

The remainder of this paper is organized as follows: Section 2 describes the sanitization problem in detail. Section 3 presents our verification methodology and results for existing hard disk-oriented techniques. Section 4 describes our FTL extensions to support single-file sanitization, and Section 5 presents our conclusions.

2 Sanitizing SSDs

The ability to reliably erase data from a storage device is critical to maintaining the security of that data. This paper identifies and develops effective methods for erasing data from solid-state drives (SSDs). Before we can address these goals, however, we must understand what it means to sanitize storage. This section establishes that definition while briefly describing techniques used to erase hard drives. Then, it explains why those techniques may not apply to SSDs.

2.1 Defining "sanitized"

In this work, we use the term "sanitize" to describe the process of erasing all or part of a storage device so that the data it contained is difficult or impossible to recover. Below we describe five different levels of sanitization storage can undergo. We will use these terms to categorize and evaluate the sanitization techniques in Sections 3 and 4.

The first level is *logical sanitization*. Data in logically sanitized storage is not recoverable via standard hardware interfaces such as standard ATA or SCSI commands. Users can logically sanitize an entire hard drive or an individual file by overwriting all or part of the drive, respectively. Logical sanitization corresponds to "clearing" as defined in NIST 800-88 [25], one of several documents from governments around the world [11, 26, 9, 13, 17, 10] that provide guidance for data destruction.

The next level is *digital sanitization*. It is not possible to recover data from digitally sanitized storage via any digital means, including undocumented drive commands or subversion of the device's controller or firmware. On disks, overwriting and then deleting a file suffices for both logical and digital sanitization with the caveat that overwriting may not digitally sanitize bad blocks that the drive has retired from use. As we shall see, the complexity of SSDs makes digitally sanitizing them more complicated.

The next level of sanitization is *analog sanitization*. Analog sanitization degrades the analog signal that encodes the data so that reconstructing the signal is effectively impossible even with the most advanced sensing equipment and expertise. NIST 800-88 refers to analog sanitization as "purging."

An alternative approach to overwriting or otherwise obliterating bits is to *cryptographically sanitize* storage. Here, the drive uses a cryptographic key to encrypt and decrypt incoming and outgoing data. To sanitize the drive, the user issues a command to sanitize the storage that holds the key. The effectiveness of cryptographic sanitization relies on the security of the encryption system used (e.g., AES [24]), and upon the designer's ability to eliminate "side channel" attacks that might allow an adversary to extract the key or otherwise bypass the encryption.

The correct choice of sanitization level for a particular application depends on the sensitivity of the data and the means and expertise of the expected adversary. Many government standards [11, 26, 9, 13, 17, 10] and secure erase software programs use multiple overwrites to erase data on hard drives. As a result many individuals and companies rely on software-based overwrite techniques for disposing of data. To our knowledge (based on working closely with several government agencies), no one has ever publicly demonstrated bulk recovery of data from an HDD after such erasure, so this confidence is probably well-placed.[1].

2.2 SSD challenges

The internals of an SSD differ in almost every respect from a hard drive, so assuming that the erasure techniques that work for hard drives will also work for SSDs is dangerous.

SSDs use flash memory to store data. Flash memory is divided into pages and blocks. Program operations apply to pages and can only change 1s to 0s. Erase operations apply to blocks and set all the bits in a block to 1. As a result, in-place update is not possible. There are typically 64-256 pages in a block (see Table 5).

A flash translation layer (FTL) [15] manages the mapping between logical block addresses (LBAs) that are visible via the ATA or SCSI interface and physical pages

[1] Of course, there may have been non-public demonstration.

of flash memory. Because of the mismatch in granularity between erase operations and program operations in flash, in-place update of the sector at an LBA is not possible.

Instead, to modify a sector, the FTL will write the new contents for the sector to another location and update the map so that the new data appears at the target LBA. As a result, the old version of the data remains in digital form in the flash memory. We refer to these “left over” data as *digital remnants*.

Since in-place updates are not possible in SSDs, the overwrite-based erasure techniques that work well for hard drives may not work properly for SSDs. Those techniques assume that overwriting a portion of the LBA space results in overwriting the same physical media that stored the original data. Overwriting data on an SSD results in logical sanitization (i.e., the data is not retrievable via the SATA or SCSI interface) but not digital sanitization.

Analog sanitization is more complex for SSDs than for hard drives as well. Gutmann [20, 19] examines the problem of data remnants in flash, DRAM, SRAM, and EEPROM, and recently, so-called “cold boot” attacks [21] recovered data from powered-down DRAM devices. The analysis in these papers suggests that verifying analog sanitization in memories is challenging because there are many mechanisms that can imprint remnant data on the devices.

The simplest of these is that the voltage level on an erased flash cell’s floating gate may vary depending on the value it held before the erase command. Multi-level cell devices (MLC), which store more than one bit per floating gate, already provide stringent control the voltage in an erased cell, and our conversations with industry [1] suggest that a single erasure may be sufficient. For devices that store a single bit per cell (SLC) a single erasure may not suffice. We do not address analog erasure further in this work.

The quantity of digital remnant data in an SSD can be quite large. The SSDs we tested contain between 6 and 25% more physical flash storage than they advertise as their logical capacity. Figure 1 demonstrates the existence of the remnants in an SSD. We created 1000 small files on an SSD, dismantled the drive, and searched for the files’ contents. The SSD contained up to 16 stale copies of some of the files. The FTL created the copies during garbage collection and out-of-place updates.

Complicating matters further, many drives encrypt data and some appear to compress data as well to improve write performance: one of our drives rumored to use compression is 25% faster for writes of highly compressible data than incompressible data. This adds an additional level of complexity not present in hard drives.

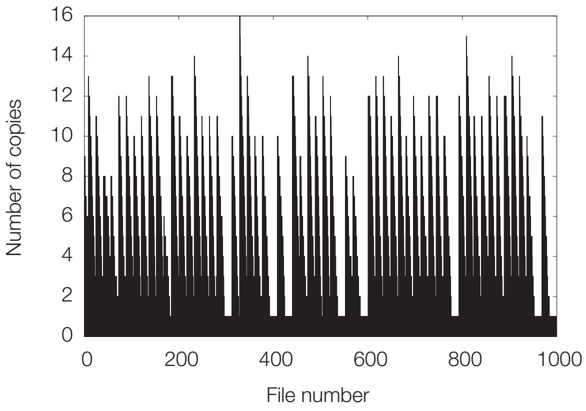


Figure 1: **Multiple copies** This graph shows The FTL duplicating files up to 16 times. The graph exhibits a spiking pattern which is probably due to the page-level management by the FTL.

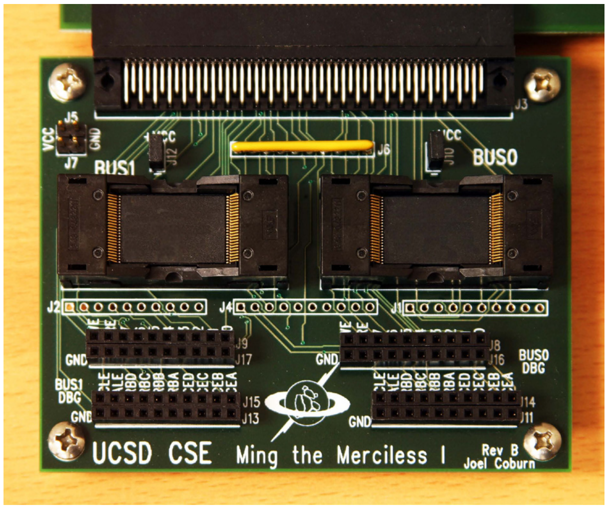


Figure 2: **Ming the Merciless** Our custom FPGA-based flash testing hardware provides direct access to flash chips without interference from an FTL.

Unless the drive is encrypted, recovering remnant data from the flash is not difficult. Figure 2 shows the FPGA-based hardware we built to extract remnants. It cost $1000 to build, but a simpler, microcontroller-based version would cost as little as $200, and would require only a moderate amount of technical skill to construct.

These differences between hard drives and SSDs potentially lead to a dangerous disconnect between user expectations and the drive’s actual behavior: An SSD’s owner might apply a hard drive-centric sanitization technique under the misguided belief that it will render the data essentially irrecoverable. In truth, data may remain on the drive and require only moderate sophistication to extract. The next section quantifies this risk by applying commonly-used hard drive-oriented techniques to SSDs and attempting to recover the “deleted” data.

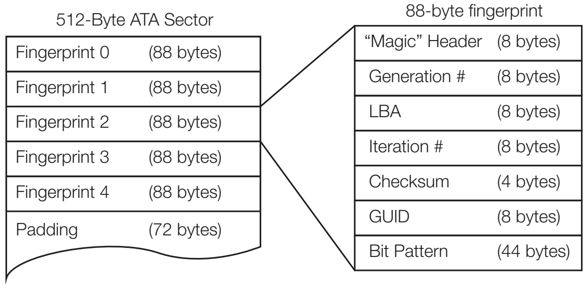


Figure 3: **Fingerprint structure** The easily-identified fingerprint simplifies the task of identifying and reconstructing remnant data.

3 Existing techniques

This section describes our procedure for testing sanitization techniques and then uses it to determine how well hard drive sanitization techniques work for SSDs. We consider both sanitizing an entire drive at once and selectively sanitizing individual files. Then we briefly discuss our findings in relation to government standards for sanitizing flash media.

3.1 Validation methodology

Our method for verifying digital sanitization operations uses the lowest-level digital interface to the data in an SSD: the pins of the individual flash chips.

To verify a sanitization operation, we write an identifiable data pattern called a *fingerprint* (Figure 3) to the SSD and then apply the sanitization technique under test. The fingerprint makes it easy to identify remnant digital data on the flash chips. It includes a sequence number that is unique across all fingerprints, byte patterns to help in identifying and reassembling fingerprints, and a checksum. It also includes an identifier that we use to identify different sets of fingerprints. For instance, all the fingerprints written as part of one overwrite pass or to a particular file will have the same identifier. Each fingerprint is 88 bytes long and repeats fives times in a 512-byte ATA sector.

Once we have applied the fingerprint and sanitized the drive, we dismantle it. We use the flash testing system in Figure 2 to extract raw data from its flash chips. The testing system uses an FPGA running a Linux software stack to provide direct access to the flash chips.

Finally, we assemble the fingerprints and analyze them to determine if the sanitization was successful. SSDs vary in how they spread and store data across flash chips: some interleave bytes between chips (e.g., odd bytes on one chip and even bytes on another) and others invert data before writing. The fingerprint's regularity makes it easy to identify and reassemble them, despite these complications. Counting the number of fingerprints that remain and categorizing them by their IDs allows us to measure the sanitization's effectiveness.

SSD	Ctlr #	SECURITY	SEC. ERASE
#	& Type	ERASE UNIT	UNIT ENH
A	1-MLC	Not Supported	Not Supported
B	2-SLC	Failed∗	Not Supported
C	1-MLC	Failed†	Not Supported
D	3-MLC	Failed†	Not Supported
E	4-MLC	Encrypted‡	Encrypted‡
F	5-MLC	Success	Success
G	6-MLC	Success	Success
H	7-MLC	Success	Success
I	8-MLC	Success	Success
J⋆	9-TLC	Not Supported	Not Supported
K⋆	10-MLC	Not Supported	Not Supported
L⋆	11-MLC	Not Supported	Not Supported

∗Drive reported success but all data remained on drive
†Sanitization only successful under certain conditions
‡Drive encrypted, unable to verify if keys were deleted
⋆USB mass storage device does not support ATA security [30]

Table 1: **Built-in ATA sanitize commands** Support for built-in ATA security commands varied among drives, and three of the drives tested did not properly execute a sanitize command it reported to support.

3.2 Whole-drive sanitization

We evaluate three different techniques for sanitizing an entire SSD: issuing a built-in sanitize command, repeatedly writing over the drive using normal IO operations, and degaussing the drive. Then we briefly discuss leveraging encryption to sanitize SSDs.

3.2.1 Built-in sanitize commands

Most modern drives have built-in sanitize commands that instruct on-board firmware to run a sanitization protocol on the drive. Since the manufacturer has full knowledge of the drive's design, these techniques should be very reliable. However, implementing these commands is optional in the drive specification standards. For instance, removable USB drives do not support them as they are not supported under the USB Mass Storage Device class [30].

The ATA security command set specifies an "ERASE UNIT" command that erases all user-accessible areas on the drive by writing all binary zeros or ones [3]. There is also an enhanced "ERASE UNIT ENH" command that writes a vendor-defined pattern (presumably because the vendor knows the best pattern to eliminate analog remnants). The new ACS-2 specification [4], which is still in draft at the time of this writing, specifies a "BLOCK ERASE" command that is part of its SANITIZE feature set. It instructs a drive to perform a block erase on all memory blocks containing user data even if they are not user-accessible.

We collected 12 different SSDs and determined if they supported the security and sanitize feature sets. If the SSD supported the command, we verified effectiveness by writing a fingerprint to the entire drive several times and then issuing the command. Overwriting several times fills as much of the over-provision area as possible with fingerprint data.

Support and implementation of the built in commands varied across vendors and firmware revisions (Table 1). Of the 12 drives we tested, none supported the ACS-2 “SANITIZE BLOCK ERASE” command. This is not surprising, since the standard is not yet final. Eight of the drives reported that they supported the ATA SECURITY feature set. One of these encrypts data, so we could not verify if the sanitization was successful. Of the remaining seven, only four executed the “ERASE UNIT” command reliably.

Drive B’s behavior is the most disturbing: it reported that sanitization was successful, but *all* the data remained intact. In fact, the filesystem was still mountable. Two more drives suffered a bug that prevented the ERASE UNIT command from working unless the drive firmware was recently reset, otherwise the command would only erase the first LBA. However, they accurately reported that the command failed.

The wide variance among the drives leads us to conclude that each implementation of the security commands must be individually tested before it can be trusted to properly sanitize the drive.

In addition to the standard commands, several drive manufacturers also provide special utilities that issue non-standard erasure commands. We did not test these commands, but we expect that results would be similar to those for the ATA commands: most would work correctly but some may be buggy. Regardless, we feel these non-standard commands are of limited use: the typical user may not know which model of SSD they own, let alone have the wherewithal to download specialized utilities for them. In addition, the usefulness of the utility depends on the manufacture keeping it up-to-date and available online. Standardized commands should work correctly almost indefinitely.

3.2.2 Overwrite techniques

The second sanitization method is to use normal IO commands to overwrite each logical block address on the drive. Repeated software overwrite is at the heart of many disk sanitization standards [11, 26, 9, 13, 17, 10] and tools [23, 8, 16, 5]. All of the standards and tools we have examined use a similar approach: They sequentially overwrite the entire drive with between 1 and 35 bit patterns. The US Air Force System Instruction 5020 [2] is typical: It first fills the drive with binary zeros, then binary ones, and finally an arbitrary character. The data is then read back to confirm that only the character is present.

SSD	Seq. 20 Pass		Rand. 20 Pass	
Init:	Seq.	Rand.	Seq.	Rand.
A	>**20**	N/A∗	N/A∗	N/A∗
B	1	N/A∗	N/A∗	N/A∗
C	2	2	2	2
D	2	2	N/A∗	N/A∗
F	2	121 hr.⋆	121 hr.⋆	121 hr.⋆
J	2	70 hr.⋆	70 hr.⋆	70 hr.⋆
K	2	140 hr.⋆	140 hr.⋆	140 hr.⋆
L	2	58 hr.⋆	58 hr.⋆	58 hr.⋆

∗Insufficient drives to perform test

⋆ Test took too long to perform, time for single pass indicated.

Table 2: **Whole-disk software overwrite.** The number in each column indicates the number of passes needed to erase data on the drive. Drives G through I encrypt, so we could not conclude anything about the success of the techniques.

The varied bit patterns aim to switch as many of the physical bits on the drive as possible and, therefore, make it more difficult to recover the data via analog means.

Bit patterns are potentially important for SSDs as well, but for different reasons. Since some SSDs compress data before storing, they will write fewer bits to the flash if the data is highly compressible. This suggests that for maximum effectiveness, SSD overwrite procedures should use random data. However, only one of the drives we tested (Drive G) appeared to use compression, and since it also encrypts data we could not verify sanitization.

Since our focus is on digital erasure, the bit patterns are not relevant for drives that store unencrypted, uncompressed data. This means we can evaluate overwrite techniques in general by simply overwriting a drive with many generations of fingerprints, extracting its contents, and counting the number of generations still present on the drive. If *k* generations remain, and the first generation is completely erased, then *k* passes are sufficient to erase the drive.

The complexity of SSD FTLs means that the usage history before the overwrite passes may impact the effectiveness of the technique. To account for this, we prepared SSDs by writing the first pass of data either sequentially or randomly. Then, we performed 20 sequential overwrites. For the random writes, we wrote every LBA exactly once, but in a pseudo-random order.

Table 2 shows the results for the eight non-encrypting drives we tested. The numbers indicate how many generations of data were necessary to erase the drive. For some drives, random writes were prohibitively slow, taking as long as 121 hours for a single pass, so we do not perform the random write test on these drives. In most cases, overwriting the entire disk twice was sufficient to sanitize the disk, regardless of the previous state of the drive. There were three exceptions: about 1% (1 GB) of the data remained on Drive A after twenty passes. We also tested a commercial implementation of the four-pass 5220.22-M standard [12] on Drive C. For the sequential initialization case, it removed all the data, but with random initialization, a single fingerprint remained. Since our testing procedure destroys the drive, we did not perform some test combinations.

Overall, the results for overwriting are poor: while overwriting appears to be effective in some cases across a wide range of drives, it is clearly not universally reliable. It seems unlikely that an individual or organization expending the effort to sanitize a device would be satisfied with this level of performance.

3.2.3 Degaussing

We also evaluated degaussing as a method for erasing SSDs. Degaussing is a fast, effective means of destroying hard drives, since it removes the disks low-level formatting (along with all the data) and damages the drive motor. The mechanism flash memories use to store data is not magnetism-based, so we did not expect the degausser to erase the flash cells directly. However, the strong alternating magnetic fields that the degausser produces will induce powerful eddy currents in chip's metal layers. These currents may damage the chips, leaving them unreadable.

We degaussed individual flash chips written with our fingerprint rather than entire SSDs. We used seven chips (marked with † in Table 5) that covered SLC, MLC and TLC (triple-level cell) devices across a range of process generation feature sizes. The degausser was a Security, Inc. HD-3D hard drive degausser that has been evaluated for the NSA and can thoroughly sanitize modern hard drives. It degaussed the chips by applying a rotating 14,000 gauss field co-planar to the chips and an 8,000 gauss perpendicular alternating field. In all cases, the data remained intact.

3.2.4 Encryption

Many recently-introduced SSDs encrypt data by default, because it provides increased security. It also provides a quick means to sanitize the device, since deleting the encryption key will, in theory, render the data on the drive irretrievable. Drive E takes this approach.

The advantage of this approach is that it is very fast: The sanitization command takes less than a second for Drive E. The danger, however, is that it relies on the controller to properly sanitize the internal storage location that holds the encryption key and any other derived values that might be useful in cryptanalysis. Given the bugs we found in some implementations of secure erase commands, it is unduly optimistic to assume that SSD vendors will properly sanitize the key store. Further, there is no way verify that erasure has occurred (e.g., by dismantling the drive).

A hybrid approach called SAFE [29] can provide both speed and verifiability. SAFE sanitizes the key store and then performs an erase on each block in a flash storage array. When the erase is finished, the drive enters a "verifiable" state. In this state, it is possible to dismantle the drive and verify that the erasure portion of the sanitization process was successful.

3.3 Single-file sanitization

Sanitizing single files while leaving the rest of the data in the drive intact is important for maintaining data security in drives that are still in use. For instance, users may wish to destroy data such as encryption keys, financial records, or legal documents when they are no longer needed. Furthermore, for systems such as personal computers and cell phone where the operating system, programs, and user data all reside on the same SSD, sanitizing single files is the only sanitization option that will leave the system in a usable state.

Erasing a file is a more delicate operation than erasing the entire drive. It requires erasing data from one or more ranges of LBAs while leaving the rest of the drive's contents untouched. Neither hard disks nor SSDs include specialized commands to erase specific regions of the drive[2].

Many software utilities [14, 5, 28, 23] attempt to sanitize individual files. All of them use the same approach as the software-based full-disk erasure tools: they overwrite the file multiple times with multiple bit patterns and then delete it. Other programs will repeatedly overwrite the free space (i.e., space that the file system has not allocated to a file) on the drive to securely erase any deleted files.

We test 13 protocols, published as a variety of government standards, as well as commercial software designed to erase single files. To reduce the number of drives needed to tests these techniques, we tested multiple techniques simultaneously on one drive. We formatted the drive under windows and filled a series of 1 GB files with different fingerprints. We then applied one erasure technique to each file, disassembled the drive, and searched for the fingerprints.

Because we applied multiple techniques to the drive at once, the techniques may interact: If the first technique leaves data behind, a later technique might overwrite it. However, the amount of data we recover from each file

[2] The ACS-2 draft standard [4] provide a "TRIM" command that informs drive that a range of LBAs is no longer in use, but this does not have any reliable effect on data security.

Overwrite operation	Data recovered	
	SSDs	USB
Filesystem delete	4.3 - 91.3%	99.4%
Gutmann [19]	0.8 - 4.3%	71.7%
Gutmann “Lite” [19]	0.02 - 8.7%	84.9%
US DoD 5220.22-M (7) [11]	0.01 - 4.1%	0.0 - 8.9%
RCMP TSSIT OPS-II [26]	0.01 - 9.0%	0.0 - 23.5%
Schneier 7 Pass [27]	1.7 - 8.0%	0.0 - 16.2%
German VSITR [9]	5.3 - 5.7%	0.0 - 9.3%
US DoD 5220.22-M (4) [11]	5.6 - 6.5%	0.0 - 11.5%
British HMG IS5 (Enh.) [14]	4.3 - 7.6%	0.0 - 34.7%
US Air Force 5020 [2]	5.8 - 7.3%	0.0 - 63.5%
US Army AR380-19 [6]	6.91 - 7.07%	1.1%
Russian GOST P50739-95 [14]	7.07 - 13.86%	1.1%
British HMG IS5 (Base.) [14]	6.3 - 58.3%	0.6%
Pseudorandom Data [14]	6.16 - 75.7%	1.1%
Mac OS X Sec. Erase Trash [5]	67.0%	9.8%

Table 3: **Single-file overwriting.** None of the protocols tested successfully sanitized the SSDs or the USB drive in all cases. The ranges represent multiple experiments with the same algorithm (see text).

Drive	Overwrites	Free Space	Recovered
C (SSD)	100×	20 MB	87%
C	100×	19,800 MB	79%
C	100× + defrag.	20 MB	86%
L (USB key)	100×	6 MB	64%
L	100×	500 MB	53%
L	100× + defrag.	6 MB	62%

Table 4: **Free space overwriting** Free space overwriting left most of the data on the drive, even with varying amounts of free space. Defragmenting the data had only a small effect on the data left over (1%).

is a lower bound on amount left after the technique completed. To moderate this effect, we ran the experiment three times, applying the techniques in different orders. One protocol, described in 1996 by Gutmann [19], includes 35 passes and had a very large effect on measurements for protocols run immediately before it, so we measured its effectiveness on its own drive.

All single-file overwrite sanitization protocols failed (Table 3): between 4% and 75% of the files' contents remained on the SATA SSDs. USB drives performed no better: between 0.57% and 84.9% of the data remained.

Next, we tried overwriting the free space on the drive. In order to simulate a used drive, we filled the drive with small (4 KB) and large files (512 KB+). Then, we deleted all the small files and overwrote the free space 100 times. Table 4 shows that regardless of the amount of free space on the drive, overwriting free space was not successful. Finally, we tried defragmenting the drive, reasoning that rearranging the files in the file system might encourage the FTL to reuse more physical storage locations. The table shows this was also ineffective.

3.4 Sanitization standards

Although many government standards provide guidance on storage sanitization, only one [25] (that we are aware of) provides guidance specifically for SSDs and that is limited to “USB Removable Disks.” Most standards, however, provide separate guidance for magnetic media and flash memory.

For magnetic media such as hard disks, the standards are consistent: overwrite the drive a number of times, execute the built-in secure erase command and destroy the drive, or degauss the drive. For flash memory, however, the standards do not agree. For example, NIST 800-88 [25] suggests overwriting the drive, Air Force System Security Instruction 5020 suggests ‘[using] the erase procedures provided by the manufacturer” [2], and the DSS Clearing & Sanitization matrix [11] suggests “perform[ing] a full chip erase per manufacturer's datasheet.”

None of these solutions are satisfactory: Our data shows that overwriting is ineffective and that the “erase procedures provided by the manufacturer” may not work properly in all cases. The final suggestion to perform a chip erase seems to apply to chips rather than drives, and it is easy to imagine it being interpreted incorrectly or applied to SSDs inappropriately. Should the user consult the chip manufacturer, the controller manufacturer, or the drive manufacturer for guidance on sanitization?

We conclude that the complexity of SSDs relative to hard drives requires that they provide built-in sanitization commands. Since our tests show that manufacturers do not always implement these commands correctly, they should be verifiable as well. Current and proposed ATA and SCSI standards provide no mechanism for verification and the current trend toward encrypting SSDs makes verification even harder.

Built-in commands for whole disk sanitization appear to be effective, if implemented correctly. However, no drives provide support for sanitizing a single file in isolation. The next section explores how an FTL might support this operation.

4 Erasing files

The software-only techniques for sanitizing a single file we evaluated in Section 3 failed because FTL complexity makes it difficult to reliably access a particular physical storage location. Circumventing this problem requires changes in the FTL. Previous work in this area [22] used encryption to support sanitizing individual files in a file system custom built for flash memory. This approach makes recovery from file system corruption difficult and it does not apply to generic SSDs.

This section describes FTL support for sanitizing arbitrary regions of an SSD's logical block address space. The extensions we describe leverage detailed measurements of flash memory characteristics. We briefly de-

Chip Name	Max Cycles	Tech Node	Cap. (Gb)	Page Size (B)	Pages /Block	Blocks /Plane	Planes /Die	Dies	Die Cap (Gb)
C-TLC16[†]	⋆	43nm	16	8192	⋆	8192	⋆	1	16
B-MLC32-4[*]	5,000	34 nm	128	4096	256	2048	2	4	32
B-MLC32-1[*]	5,000	34 nm	32	4096	256	2048	2	1	32
F-MLC16[*]	5,000	41 nm	16	4096	128	2048	2	1	16
A-MLC16[*]	10,000	⋆	16	4096	128	2048	2	1	16
B-MLC16[*]	10,000	50 nm	32	4096	128	2048	2	2	16
C-MLC16[†]	⋆	⋆	32	4096	⋆	⋆	⋆	2	16
D-MLC16[*]	10,000	⋆	32	4096	128	4096	1	2	16
E-MLC16[†*]	TBD	⋆	64	4096	128	2048	2	4	16
B-MLC8[*]	10,000	72 nm	8	2048	128	4096	1	1	8
E-MLC4[*]	10,000	⋆	8	4096	128	1024	1	2	4
E-SLC8[†*]	100,000	⋆	16	4096	64	2048	2	2	8
A-SLC8[*]	100,000	⋆	8	2048	64	4096	2	1	8
A-SLC4[*]	100,000	⋆	4	2048	64	4096	1	1	4
B-SLC2[*]	100,000	50 nm	2	2048	64	2048	1	1	2
B-SLC4[*]	100,000	72 nm	4	2048	64	2048	2	1	4
E-SLC4[*]	100,000	⋆	8	2048	64	4096	1	2	4
A-SLC2[*]	100,000	⋆	2	2048	64	1024	2	1	2

[*] Chips tested for data scrubbing. [†] Chips tested for degaussing. ⋆ No data available

Table 5: **Flash Chip Parameters.** Each name encodes the manufacturer, cell type and die capacity in Gbits. Parameters are drawn from datasheets where available. We studied 18 chips from 6 manufacturers.

scribe our baseline FTL and the details of flash behavior that our technique relies upon. Then, we present and evaluate three ways an FTL can support single-file sanitization.

4.1 The flash translation layer

We use the FTL described in [7] as a starting point. The FTL is page-based, which means that LBAs map to individual pages rather than blocks. It uses log-structured writes, filling up one block with write data as it arrives, before moving on to another. As it writes new data for an LBA, the old version of the data becomes invalid but remains in the array (i.e., it becomes remnant data).

When a block is full, the FTL must locate a new, erased block to continue writing. It keeps a pool of erased blocks for this purpose. If the FTL starts to run short of erased blocks, further incoming accesses will stall while it performs garbage collection by consolidating valid data and freeing up additional blocks. Once its supply of empty blocks is replenished, it resumes processing requests. During idle periods, it performs garbage collection in the background, so blocking is rarely needed.

To rebuild the map on startup, the FTL stores a reverse map (from physical address to LBA) in a distributed fashion. When the FTL writes data to a page, the FTL writes the corresponding LBA to the page's out-of-band section. To accelerate the start-up scan, the FTL stores a summary of this information for the entire block in the block's last page. This complete reverse map will also enable efficiently locating all copies of an LBA's data in our scan-based scrub technique (See Section 4.4).

4.2 Scrubbing LBAs

Sanitizing an individual LBA is difficult because the flash page it resides in may be part of a block that contains useful data. Since flash only supports erasure at the block level, it is not possible to erase the LBA's contents in isolation without incurring the high cost of copying the entire contents of the block (except the page containing the target LBA) and erasing it.

However, *programming* individual pages is possible, so an alternative would be to re-program the page to turn all the remaining 1s into 0s. We call this *scrubbing* the page. A scrubbing FTL could remove remnant data by scrubbing pages that contain stale copies of data in the flash array, or it could prevent their creation by scrubbing the page that contained the previous version whenever it wrote a new one.

The catch with scrubbing is that manufacturer

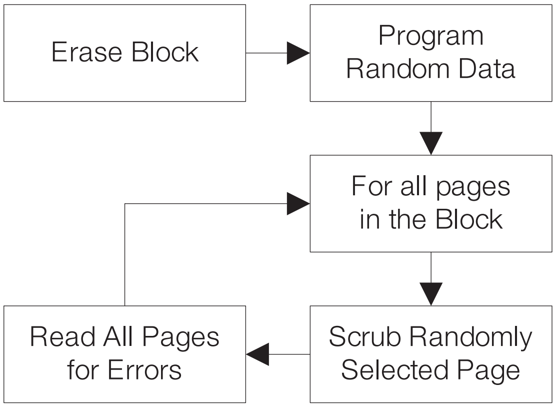

Erase Block
Program Random Data
For all pages in the Block
Scrub Randomly Selected Page
Read All Pages for Errors

Figure 4: **Testing data scrubbing** To determine whether flash devices can support scrubbing we programmed them with random data, randomly scrubbed pages one at a time, and then checked for errors.

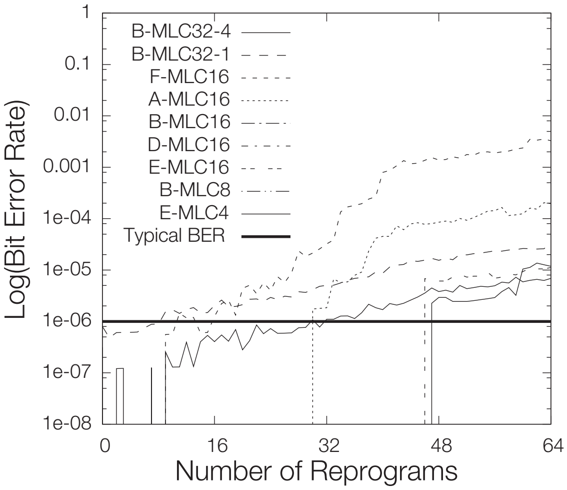


Figure 5: **Behavior under data scrubbing** Scrubbing causes more errors in some chips than others, resulting in wide variation of scrub budgets for MLC devices.

datasheets require programming the pages within a block in order to reduce the impact of program disturb effects that can increase error rates. Scrubbing would violate this requirement. However, previous work [18] shows that the impact of reprogramming varies widely between pages and between flash devices, and that, in some cases, reprogramming (or scrubbing) pages would have no effect.

To test this hypothesis, we use our flash testing board to scrub pages on 16 of the chips in Table 5 and measure the impact on error rate. The chips span six manufacturers, five technology nodes and include both MLC and SLC chips.

Figure 4 describes the test we ran. First, we erase the block and program random data to each of its pages to represent user data. Then, we scrub the pages in random order. After each scrub we read all pages in the block to check for errors. Flash blocks are independent, so checking for errors only within the block is sufficient. We repeated the test across 16 blocks spread across each chip.

The results showed that, for SLC devices, scrubbing did not cause any errors at all. This means that the number scrubs that are acceptable – the *scrub budget* – for SLC chips is equal to the number of pages in a block.

For MLC devices determining the scrub budget is more complicated. First, scrubbing one page invariably caused severe corruption in exactly one other page. This occurred because each transistor in an MLC array holds two bits that belong to different pages, and scrubbing one page reliably corrupts the other. Fortunately, it is easy to determine the paired page layout in all the chips we have tested, and the location of the paired page of a given page is fixed for a particular chip model. The paired page effect means that the FTL must scrub both pages in a pair at the same time, relocating the data in the page that was not the primary target of the scrub.

Figure 5 shows bit error rates for MLC devices as a function of scrub count, but excluding errors in paired pages. The data show that for three of the nine chips we tested, scrubbing caused errors in the unscrubbed data in the block. For five of the remaining devices errors start to appear after between 2 and 46 scrubs. The final chip, B-MLC32-1, showed errors without any scrubbing. For all the chips that showed errors, error rates increase steeply with more scrubbing (the vertical axis is a log scale).

It may be possible to reduce the impact of scrubbing (and, therefore, increase the scrub budget) by carefully measuring the location of errors caused by scrubbing a particular page. Program disturb effects are strongest between physically adjacent cells, so the distribution of scrubs should affect the errors they cause. As a result, whether scrubbing page is safe would depend on which other pages the FTL has scrubbed in the block, not the number of scrubs.

The data in the figure also show that denser flash devices are less amenable to scrubbing. The chips that showed no errors (B-MLC16, D-MLC16, and B-MLC8) are 50 nm or 70 nm devices, while the chips with the lowest scrub budgets (F-MLC16, B-MLC32-4, and B-MLC32-1) are 34 or 41 nm devices.

4.3 Sanitizing files in the FTL

The next step is to use scrubbing to add file sanitization support to our FTL. We consider three different methods that make different trade-offs between performance and data security – immediate scrubbing, background scrubbing, and scan-based scrubbing.

Name	Total Accesses	Reads	Description
Patch	64 GB	83%	Applies patches to the Linux kernel from version 2.6.0 to 2.6.29
OLTP	34 GB	80%	Real-time processing of SQL transactions
Berkeley-DB Btree	34 GB	34%	Transactional updates to a B+tree key/value store
Financial	17 GB	15%	Live OLTP trace for financial transactions.
Build	5.5 GB	94%	Compilation of the Linux 2.6 kernel
Software devel.	1.1 GB	65%	24 hour trace of a software development work station.
Swap	800 MB	84%	Virtual memory trace for desktop applications.

Table 6: **Benchmark and application traces** We use traces from eight benchmarks and workloads to evaluate scrubbing.

These methods will eliminate all remnants in the drive's spare area (i.e., that are not reachable via a logical block address). As a result, if a file system does not create remnants on a normal hard drive (e.g., if the file system overwrite a file's LBAs when it performs a delete), it will not create remnants when running on our FTL.

Immediate scrubbing provides the highest level of security: write operations do not complete until the scrubbing is finished – that is, until FTL has scrubbed the page that contained the old version of the LBA's contents. In most cases, the performance impact will be minimal because the FTL can perform the scrub and the program in parallel.

When the FTL exceeds the scrub budget for a block, it must copy the contents of the block's valid pages to a new block and then erase the block before the operation can complete. As a result, small scrub budgets (as we saw for some MLC devices) can degrade performance. We measure this effect below.

Background scrubbing provides better performance by allowing writes to complete and then performing the scrubbing in the background. This results in a brief window when remnant data remains on the drive. Background scrubbing can still degrade performance because the scrub operations will compete with other requests for access to the flash.

Scan-based scrubbing incurs no performance overhead on normal write operations but adds a command to sanitize a range of LBAs by overwriting the current contents of the LBAs with zero and then scrubbing any storage that previously held data for the LBAs. This technique exploits the reverse (physical to logical) address map that the SSD stores to reconstruct the logical-to-physical map.

To execute a scan-based scrubbing command, the FTL reads the summary page from each block and checks if any of the pages in the block hold a copy of an LBA that the scrub command targets. If it does, the FTL scrubs that page. If it exceeds the scrub budget, the FTL will need to relocate the block's contents.

We also considered an SSD command that would apply scrubbing to specific write operations that the operating system or file system marked as "sanitizing." However, immediate and background scrubbing work by guaranteeing that only one valid copy of an LBA exists by always scrubbing old version when writing the new version. Applying scrubbing to only a subset of writes would violate this invariant and allow the creation of remnants that a single scrub could not remove.

4.4 Results

To understand the performance impact of our scrubbing techniques, we implemented them in a trace-based FTL simulator. The simulator implements the baseline FTL described above and includes detailed modeling of command latencies (based on measurements of the chips in Table 5) and garbage collection overheads. For these experiments we used E-SLC8 to collect SLC data and F-MLC16 to for MLC data. We simulate a small, 16 GB SSD with 15% spare area to ensure that the FTL does frequent garbage collection even on the shorter traces.

Table 6 summarizes the eight traces we used in our experiments. They cover a wide range of applications from web-based services to software development to databases. We ran each trace on our simulator and report the latency of each FTL-level page-sized access and trace run time. Since the traces include information about when each the application performed each IO, the change in trace run-time corresponds to application-level performance changes.

Immediate and background scrubbing Figure 6 compares the write latency for immediate and background scrubbing on SLC and MLC devices. For MLC, we varied the number of scrubs allowed before the FTL must copy out the contents of the block. The figure normalizes the data to the baseline configuration that does not perform scrubbing or provide any protection against remnant data.

For SLC-based SSDs, immediate scrubbing causes no decrease in performance, because scrubs frequently execute in parallel with the normal write access.

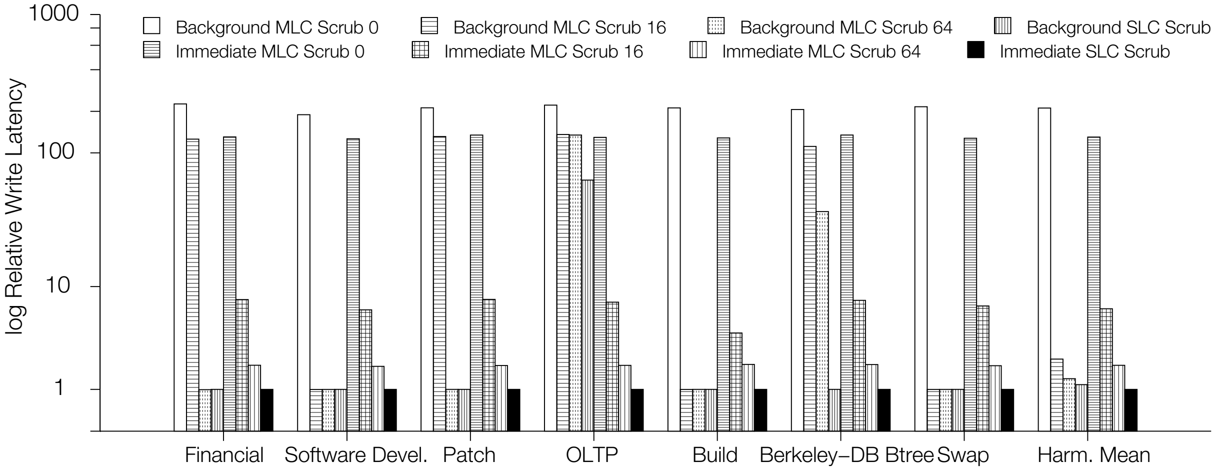


Figure 6: **Immediate and background scrubbing performance** For chips that can withstand at least 64 scrub operations, both background and immediate scrubbing can prevent the creation of data remnants with minimal performance impact. For SLC devices (which can support unlimited scrubbing), background scrubbing has almost no effect and immediate scrubbing increases write latency by about 2×.

In MLC devices, the cost of immediate scrubbing can be very high if the chip can tolerate only a few scrubs before an erase. For 16 scrubs, operation latency increases by 6.4× on average and total runtime increases by up to 11.0×, depending on the application. For 64 scrubs, the cost drops to 2.0× and 3.2×, respectively.

However, even a small scrub budget reduces latency significantly compared relying on using erases (and the associated copy operations) to prevent remnants. Tmplementing immediate sanitization with just erase commands increases operation latency by 130× on average (as shown by the "Scrub 0" data in Figure 5).

If the application allows time for background operations (e.g., Build, Swap and Dev), background scrubbing with a scrub budget of 16 or 64 has a negligible effect on performance. However, when the application issues many requests in quick succession (e.g., OLTP and BDB), scrubbing in the background strains the garbage collection system and write latencies increase by 126× for 16 scrubs and 85× for 64 scrubs. In contrast, slowdown for immediate scrubbing range from just 1.9 to 2.0× for a scrub budget of 64 and from 4.1 to 7.9× for 16 scrubs.

Scrubbing also increases the number of erases required and, therefore, speeds up program/erase-induced wear out. Our results for MLC devices show that scrubbing increased wear by 5.1× for 16 scrubs per block and 2.0× with 64 scrubs per block. Depending on the application, the increased wear for chips that can tolerate only a few scrubs may or may not be acceptable. Scrubbing SLC devices does not require additional erase operations.

Finally, scrubbing may impact the long-term integrity of data stored in the SSD in two ways. First, although manufactures guarantee that data in brand new flash devices will remain intact for at least 10 years, as the chip ages data retention time drops. As a result, the increase in wear that scrubbing causes will reduce data retention time over the lifetime of the SSD. Second, even when scrubbing does not cause errors immediately, it may affect the analog state of other cells, making it more likely that they give rise to errors later. Figure 6 demonstrates the analog nature of the effect: B-MLC32-4 shows errors that come and go for eight scrubs.

Overall, both immediate and background scrubbing are useful options for SLC-based SSDs and for MLC-based drives that can tolerate at least 64 scrubs per block. For smaller scrub budgets, both the increase in wear and the increase in write latency make these techniques costly. Below, we describe another approach to sanitizing files that does not incur these costs.

Scan-based scrubbing Figure 7 measures the latency for a scan-based scrubbing operation in our FTL. We ran each trace to completion and then issued a scrub command to 1 GB worth of LBAs from the middle of the device. The amount of scrubbing that the chips can tolerate affects performance here as well: scrubbing can reduce the scan time by as much as 47%. However, even for the case where we must use only erase commands (MLC-scrub-0), the operation takes a maximum of 22 seconds. This latency breaks down into two parts – the time required to scan the summary pages in each block (0.64 s

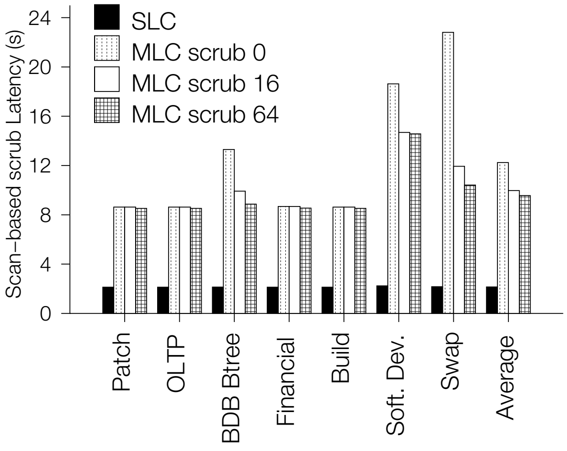


Figure 7: **Scan-based scrubbing latency** The time to scrub 1 GB varies with the number of scrubs each block can withstand, but in all cases the operation takes less than 30 seconds.

for our SLC SSD and 1.3 s for MLC) and the time to perform the scrubbing operations and the resulting garbage collection. The summary scan time will scale with SSD size, but the scrubbing and garbage collection time are primarily a function of the size of the target LBA region. As a result, scan-based scrubbing even on large drives will be quick (e.g., ∼62 s for a 512 GB drive).

5 Conclusion

Sanitizing storage media to reliably destroy data is an essential aspect of overall data security. We have empirically measured the effectiveness of hard drive-centric sanitization techniques on flash-based SSDs. For sanitizing entire disks, built-in sanitize commands are effective when implemented correctly, and software techniques work most, but not all, of the time. We found that none of the available software techniques for sanitizing individual files were effective. To remedy this problem, we described and evaluated three simple extensions to an existing FTL that make file sanitization fast and effective. Overall, we conclude that the increased complexity of SSDs relative to hard drives requires that SSDs provide verifiable sanitization operations.

Acknowledgements

The authors would like to thank Jack Sampson for his useful comments. This work was supported in part by NSF award 1029783.

References

[1] M. Abraham. NAND flash security. In *Special Pre-Conference Workshop on Flash Security*, August 2010.

[2] U. S. Air Force. *Air Force System Security Instruction 5020*, 1998.

[3] American National Standard of Accredited Standards Committee X3T13. *Information Technology - AT Attachment-3 Interface*, January 1997.

[4] American National Standard of Accredited Standards INCITS T13. *Information Technology - ATA/ATAPI Command Set - 2*, June 2010.

[5] Apple Inc. Mac OS X 10.6, 2009.

[6] U. S. Army. *Army Regulation 380-19*, 1998.

[7] A. Birrell, M. Isard, C. Thacker, and T. Wobber. A design for high-performance flash disks. Technical Report MSR-TR-2005-176, Microsoft Research, December 2005.

[8] Blancco Oy Ltd. Blancco PC Edition 4.10.1. http://www.blancco.com.

[9] Bundesamts fr Sicherheit in der Informationstechnik. *Richtlinien zum Geheimschutz von Verschlusssachen beim Einsatz von Informationstechnik*, December 2004.

[10] A. Defence Signals Directorate. *Government Information Security Manual (ISM)*, 2006.

[11] U. S. Defense Security Services. *Clearing and Sanitization Matrix*, June 2007.

[12] U. S. Department of Defense. *5220.22-M National Industrial Security Program Operating Manual*, January 1995.

[13] U. S. Dept. of the Navy. *NAVSO P-5239-08 Network Security Officer Guidebook*, March 1996.

[14] Eraser. http://eraser.heidi.ie/.

[15] E. Gal and S. Toledo. Algorithms and data structures for flash memories. *ACM Comput. Surv.*, 37(2):138–163, 2005.

[16] GEEP EDS LLC. Darik's Boot and Nuke ("DBAN"). http://www.dban.org/.

[17] N. Z. Government Communications Security Bureau. *Security of Information NZSIT 402*, Feburary 2008.

[18] 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 *MICRO'09: Proceedings of ...*, New York, NY, USA, 2009. ACM, IEEE.

[19] P. Gutmann. Secure deletion of data from magnetic and solid-state memory. In *SSYM'96: Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography*, pages 8–8, Berkeley, CA, USA, 1996. USENIX Association.

[20] P. Gutmann. Data remanence in semiconductor devices. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 4–4, Berkeley, CA, USA, 2001. USENIX Association.

[21] J. A. Halderman, S. D. Schoen, N. Heninger,

W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: cold-boot attacks on encryption keys. *Commun. ACM*, 52(5):91–98, 2009.

[22] J. Lee, J. Heo, Y. Cho, J. Hong, and S. Y. Shin. Secure deletion for nand flash file system. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 1710–1714, New York, NY, USA, 2008. ACM.

[23] LSoft Technologies Inc. Active@ KillDisk. http://www.killdisk.com/.

[24] U. S. National Institute of Standards and Technology. *Advanced Encryption Standard (AES) (FIPS PUB 197)*, November 2001.

[25] U. S. National Institute of Standards and Technology. *Special Publication 800-88: Guidelines for Media Sanitization*, September 2006.

[26] Royal Canadian Mounted Police. *G2-003, Hard Drive Secure Information Removal and Destruction Guidelines*, October 2003.

[27] B. Schneier. *Applied cryptography (2nd ed.): protocols, algorithms, and source code in C.* John Wiley & Sons, Inc., New York, NY, USA, 1995.

[28] secure rm. http://srm.sourceforge.net/.

[29] S. Swanson and M. Wei. Safe: Fast, verifiable sanitization for ssds. http://nvsl.ucsd.edu/sanitize/, October 2010.

[30] USB Implementers Forum. *Universal Serial Bus Mass Storage Class Specification Overview*, September 2008.

A Scheduling Framework that Makes any Disk Schedulers Non-work-conserving solely based on Request Characteristics

Yuehai Xu
ECE Department
Wayne State University
Detroit, MI 48202, USA
yhxu@wayne.edu

Song Jiang
ECE Department
Wayne State University
Detroit, MI 48202, USA
sjiang@eng.wayne.edu

Abstract

Exploiting spatial locality is critical for a disk scheduler to achieve high throughput. Because of the high cost of disk head seeks and the non-preemptible nature of request service, state-of-the-art disk schedulers consider the locality of both pending and future requests. Though schedulers adopting the approach, such as the anticipatory scheduler, show substantial performance advantages, they need to know from which processes requests are issued to evaluate locality. This approach is not effective when the knowledge about processes is not available (e.g., in virtual machine environment, network or parallel file systems, and SAN) or the locality exhibited on a disk region is not solely determined by individual processes (e.g., in the case of cooperative process groups and disk array where requested data are striped).

We propose a light-weight disk scheduling framework that does not require any process knowledge for analyzing request locality. Solely based on requests' own characteristics the framework can make any work-conserving scheduler non-work-conserving, i.e., able to take future requests as dispatching candidates, to fully exploit locality. Additionally, we show how to effectively extend the framework to the disk array environment. Our design, *Stream Scheduling*, is prototyped in the Linux kernel 2.6.31. With extensive experiments of representative benchmarks, and in various environments such as the Xen virtual machine and the PVFS parallel file system, we show that the proposed scheduling framework can improve their performance by up to 3.2 times.

1 Introduction

While the hard disk has maintained exponential growth in capacity as a function of time, and sustained improvement in peak throughput, its random access performance, which is mainly determined by disk seek time, is increasingly a bottleneck. This makes the disk scheduler, which aims to minimize disk seeks by exploiting spatial locality in the requests, increasingly important to disk performance.

1.1 Non-work-conserving Disk Scheduling

Traditionally a disk scheduler such as CSCAN and SPTF chooses a request from those that have arrived and are pending in its dispatch queue and dispatches it to the disk. In a *work-conserving* mode, the scheduler must choose one of the pending requests, if any, to dispatch, even if the pending requests are far away from the current disk head position. The rationale for *non-work-conserving* schedulers, such as the anticipatory scheduler (AS) [16] and Completely Fair Queuing (CFQ) [1], is that a request that is soon to arrive might be much closer to the disk head than the currently pending requests, in which case it may be worthwhile to wait for the future request.[1] If such a request does arrive soon and the benefit of avoiding the long-distance disk seek outweighs the cost of idle waiting, the decision to keep the disk head in place may be justified. This is commonly observed when there are multiple processes concurrently issuing synchronous requests. For a request synchronously issued by a process, the scheduler can see its next request only after the request is served. Without a short waiting period the spatial locality of requests from such a process cannot be exploited. In this context the spatial locality refers to the fact that nearby disk locations are likely to be accessed by two consecutive requests within a short period of time. A process has strong locality if soon after its current request is completed, the scheduler will receive its next request for a location close to the current request. While the traditional scheduler selects a request for dispatching only from currently pending requests, a non-work-conserving scheduler, in essence, selects one from currently pending requests *and* future requests to exploit locality among synchronously issued requests.

[1] Descriptions of requests' statuses, such as "currently pending" or "future requests", are relative to the time when a scheduling decision is being made.

1.2 The Issues

To be effective, a non-work-conserving scheduler needs to predict how long it will take for the next nearby request to arrive—the strength of the process's locality—with reasonable accuracy, so that a decision can be whether to wait, and if so, for how long. To this end, existing non-work-conserving schedulers, such as AS and CFQ, group requests according to their issuing processes, analyze locality for each group, and make predictions for each process. While analyzing and utilizing locality in the context of process is an intuitive and convenient choice, there are three scenarios that challenge this practice.

First, if the requests to a limited disk region are from multiple processes, the locality, which is the basis for any scheduler to make scheduling decisions, is the result of these processes' combined I/O behaviors. This is especially the case when these processes coordinate to issue their requests. To determine whether the disk head should wait for a future request, the scheduler cares only about the probability for a nearby request to appear quickly, regardless of whether the request is from the same process. Limiting locality analysis to each individual process may underestimate the locality actually available to the scheduler and lose opportunity for seek reduction.

Second, in many important system settings process information is not available to the disk scheduler. For example, in the virtual machine environment only the scheduler in the host OS or VMM can actually dispatch I/O requests to the disk, on behalf of guest VMs where processes run and generate the requests. The scheduler in the host usually can only tell from which VM it receives a request but cannot distinguish from which process on a VM the request is issued. When there are multiple processes running on a VM, lack of such knowledge at the host would make non-work-conserving host scheduler less effective. In distributed or parallel file systems such as NFS and PVFS, the daemon at the file server receives requests from the clients and passes them to the disk scheduler without telling it which processes at the client side actually issued them. For another example, the SAN system and hardware RAID have internal disk schedulers that are critical to the systems' efficiency. The system interface for through which I/O requests are accepted usually does not include process information about request source.

Third, one of assumptions made by non-work-conserving schedulers is that it is solely the process that determines how long it will take for its next request to be issued. For this reason, *thinktime*, the time period between two consecutive I/O calls of a process, is treated as an attribute of the process and is estimated using the process's history information to predict when its next request will arrive. However, if the disk is a member of a disk array over which data are striped, the next several requests from the process might go to other disks in the array and may not be immediately scheduled for those disks. Consequently, the timing for this disk to see its next request from the process is determined not only by the process's thinktimes, but also by the data striping pattern on the array as well as the scheduling decisions made at the other disks. By mistaking the time period between two consecutive requests from a process for the process's thinktime, a disk's scheduler finds little opportunity for non-work-conserving scheduling. However, the fact is that by coordinating the scheduling of disks in the array, it is possible to reduce the time period so that waiting for the next request can still be beneficial.

1.3 The Challenges

To address these issues, we have to give up the assumption on the availability of process information. Specially, a scheduler is still expected to take future requests into account when making scheduling decisions, even without the process information, so that the most suitable request among both currently pending requests and future requests can be selected for dispatching. There are several critical challenges in achieving this objective.

First, if locality were to be explicitly analyzed for predicting timing and location of the next request, we have to group requests according to some criteria to track locality for each group of requests. However, without process information, for any artificial grouping method it would be hard to accurately predict whether a request would appear whose locality is stronger than any of currently pending requests. For example, a seemingly effective method is to divide the disk into different regions, either evenly or accordingly to request concentrations, and then track locality in each region. However, if the region were set too small, one process's synchronous requests could span multiple regions, which makes the arrival of the next request in a region too late and thus the locality in each region too weak. If the region were set too large, requests in a large disk area would be included for locality tracking, making the measured locality weak because of large inter-request distance. In both cases the scheduler may lose the opportunity to schedule future requests. In addition, region size may have to be dynamically adjusted according to changing request distribution on disk, making meaningful locality analysis yet more difficult.

Second, locality is relative. When there are pending requests relatively close to the current disk head, the scheduler must evaluate only the probability of requests of strong locality, and the relatively remote requests become less relevant. In contrast, if pending requests are relatively remote, even some not-very-close requests need to be included for locality analysis so as not to lose opportunity for higher disk efficiency. Therefore one must determine which requests should be included in an analysis adapting to the locations of pending requests. This would significantly add to the complexity and cost of such algorithms.

Third, for data striped on a disk array, even if think-times can be sufficiently short for I/O-intensive applications, the time gaps between two continuous requests seen at each disk can be too large to be exploited by non-work-conserving schedulers at individual disks. In this case the challenge is whether it is possible to reduce the time gaps by coordinating individual disks' scheduling so that it becomes worthwhile for a disk to wait for a future request. If the answer is yes, the question is how to know when there is such a potential before taking action for the coordination. As such an action usually entails postponing service of other applications' requests, it could cause excessive overhead and adversely affect performance if it did not produce the expected saving in disk seek time.

1.4 Our Contributions

In this paper we propose a light-weight framework that uses only requests' characteristics, specifically requests' arrival times and requested data locations, to turn any work-conserving scheduler into a non-work-conserving one. These request characteristics are readily available in any storage system and are employed in almost all disk schedulers. In summary, we make the following contributions.

First, instead of using the conventional method of direct analysis of locality to make a prediction about future requests, we propose to track the judicious actions, either waiting for future requests or seeking to a pending request, that should have been taken for greater disk efficiency. A judicious action is the one that helps improve disk efficiency, and may or may not have actually been taken in the prior scheduling. After observing a consistent pattern of judicious actions, our scheduling framework guides the scheduler to follow the trend in making its next decision. In the meantime, the framework retains the mechanism provided by the corresponding work-conserving scheduler for avoiding long delay or even starvation in its request service. The framework is simple, efficient, effective, and minimally intrusive to the work-conserving scheduler.

Second, we propose an efficient scheme for non-work-conserving scheduling for the disk array. To this end, we create a virtual disk corresponding to a disk array and apply our proposed framework on it to evaluate the potential benefit of coordinating scheduling across the disks for a particular stream of requests. When the evaluation is positive, coordinated scheduling of all disks is conducted to make it possible for scheduling of future requests to be profitable.

Third, we have implemented and evaluated the scheduling framework for single disks and for disk arrays, collectively named *stream scheduling*, in the Linux 2.6.31 and Linux software RAID MD. Our experiments on the prototype system with a variety of benchmarks demonstrate its significant performance advantages.

Section 2 of this paper details the design of stream scheduling. Section 3 presents an extensive experimental evaluation. Section 4 describes related work, and Section 5 concludes.

2 The design of *Stream Scheduling*

While a non-work-conserving scheduler is designed to select one request of the lowest cost from currently pending requests and future requests, a key technique in the scheduling is the effective comparison of costs for serving these two types of requests. Because future requests are not available for immediate dispatching, the scheduler keeps the disk idle for some period of time waiting for them if it decides to schedule a future request. Accordingly the cost for dispatching a future request is the sum of the wait time and the request's service time, while the cost of dispatching a pending request is just its service time. To effectively implement a non-work-conserving scheduler, there are two critical questions to answer: (1) how likely it is to see a future request whose cost is lower than that of the pending requests; and, (2) which future requests can be the candidates for selection. The answer to the first question determines whether a future request should be selected—whether the disk should wait—and the answer to the second question determines the threshold of the wait period beyond which no requests would be qualified. In the proposed framework it is the stream scheduling algorithm that answers the two questions by taking three inputs, namely request arrival time, arriving request location, and pending request location.

When a scheduler is ready to dispatch a new request the stream scheduling algorithm makes the decision on whether or not to schedule a future request. If yes, it will leave the disk waiting for an incoming request of relatively strong locality. Otherwise, it will dispatch a pending request selected by the working-conserving scheduling algorithm. As the stream scheduling algorithm makes its decisions independently of the working-conserving scheduling algorithm, the scheduling framework is applicable to any working-conserving scheduling algorithms.

2.1 The Stream Scheduling Algorithm

We consider a decision to make the disk wait for future requests a judicious one if there exists a future request R such that $wait\_time(R) + service\_time(R) < service\_time(selected\_pending\_request)$, where $wait\_time(R)$ is the time period from the time when the decision is made to the time when request R arrives, $service\_time(R)$ is the time spent to serve request R, the first dispatched future request after the decision is made, and $service\_time(selected\_pending\_request)$ is the service time for request selected by the work-conserving scheduling algorithm when the decision is made. If the inequality does not hold, the decision that demands immediate dispatching of a pending request is a judicious one. Note that the evaluation of the inequality cannot be completed until a future request satisfying the inequality actually arrives or until $wait\_time(R) \geq service\_time(selected\_pending\_request)$ becomes true. To evaluate the inequality, the service time of a known request can be estimated according to the distance between the location of its requested data and current disk head position, which can be considered to be the location of the most recently served request [14, 16]. Therefore, no matter whether request $selected\_pending\_request$ is actually dispatched, $service\_time(selected\_pending\_request)$ can be estimated.

In the inequality only $service\_time(selected\_pending\_request)$ is known when the decision is being made, while $wait\_time(R)$ and $service\_time(R)$ are unknown. Generally there are two methods to predict whether the inequality will hold. One is the method adopted by existing non-work-conserving schedulers, which use wait times and service times of previous requests that belong to the same process to predict these two times for the next request from the process, respectively. This method does not work when the process information is unavailable, because we do not know which previous requests and which future requests should be included in the evaluation of the inequality. To address the issue we propose the second method, which identifies a series of recently served requests for which the inequality held to form a so-called *stream*. A stream of sufficient size indicates that it is likely that the inequality would continue to hold and a judicious decision is to wait for future requests.

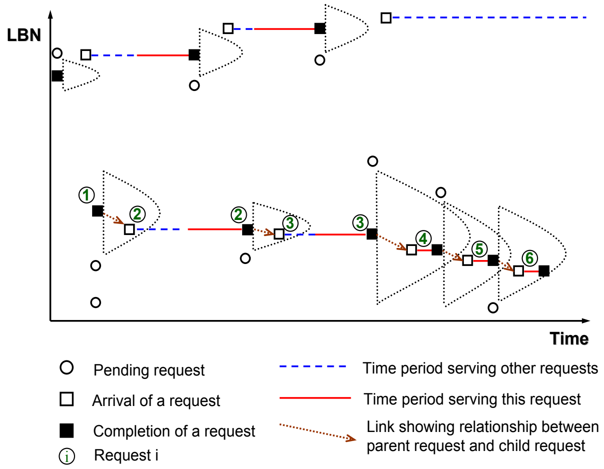

LBN
Time
1
2
2
3
3
4
5
6
Pending request
Arrival of a request
Completion of a request
Request i
Time period serving other requests
Time period serving this request
Link showing relationship between parent request and child request

Figure 1: Illustration of forming a stream and using the stream for scheduling. In the figure, the mushroom-shaped area ahead of each completed request describes the inequality on the eligibility of being a child request. The size of an area is determined by how close its corresponding pending requests are from the completed request. When a new request arrives in such an area, it becomes the child of the completed request associated with the area and extends the corresponding stream. As shown in the graph, the arrival of request 2 in the area following request 1 extends the stream to [1, 2]. When request 2 is completed, its area is created and the arrival of request 3 in the area further extends the stream to [1, 2, 3]. A stream cannot be established without new requests arriving in the defined areas, as shown in the upper part of the figure. In the lower part of the figure, before the stream is established, the disk head must leave and then seek back to serve its next request. When request 4 becomes a child request and joins the the stream, the stream is established (assuming that *stream_threshold* is 4). After this, the disk keeps serving requests in the stream (such as requests 5 and 6) for some period of time for high I/O efficiency.

Figure 1 illustrates how a stream is formed and how it is used for request scheduling. The figure shows the arrival and completion times of requests as well as the requests' positions on the disk in terms of their requested data's LBNs (Logical Block Numbers). When the scheduler is notified that a request is completed is the time for the scheduler to select one request from currently pending requests and eligible future requests, or requests satisfying the inequality. As we can see, the positions of pending requests determine the eligibility of future requests. This is what we expect. If there are nearby pending requests, the criteria to schedule a future request must be more strict to make it profitable. Otherwise, it may be affordable for the disk to wait for a longer time and/or for a request with longer distance to the recently completed request. We may not come to a conclusion on whether a future request should be selected, or whether the scheduling decision is judicious, until $service\_time(selected\_pending\_request)$ after the decision is made. Note that the conclusion does not depend on what the actual decision is. If later on we do find a request arriving at a time and a position that satisfy the inequality, this request is called the *child* of the recently completed request. Therefore, for a request that is highly likely to have a child, the scheduler should wait for the child request, instead of immediately dispatching a pending request. To predict whether a recently completed request would have a child, we introduce the concept of *stream*, which is a sequence of requests $[R_0, R_1, ..., R_{n-1}]$ that have arrived in time-ascending order. For any two adjacent requests (R_{k-1}, R_k) in the stream, R_k is the child of R_{k-1}. If the length of the stream is equal to or greater than a predefined threshold *stream_threshold*, the stream is considered *established*.

The assumption we make in the stream scheduling algorithm is that for an established stream $[R_0, R_1, ..., R_{n-1}]$ $(n \geq stream\_threshold)$, request R_{n-1} is highly likely to have its child request R_n extend the stream. The child request is the first one that arrives after the completion of R_{n-1} and satisfies the inequality, and the the disk should wait for the child request. This assumption is consistent with those made by other non-work-conserving algorithms to estimate thinktime and seek time of a process's next request. In addition, as we do not independently predict these two times, we can take the relationship between pending requests and future requests into account in the assumption. A disk waiting for a child request will stay idle for at most *service_time(selected_pending_request)* if there exist pending requests. The time when *service_time(selected_pending_request)* passes a request's completion time is called the request's deadline. After its deadline, it is not possible to find an eligible request to be the request's child. If the most recent request in a stream fails to find its child request, the stream aborts. Pseudo code for the algorithm is shown in Figure 2.

As shown in the pseudo code, when a request is completed it is possible for it to become a parent of a future request. So we insert the request into the *parent-to-be* queue to see if it would have a child that turns it into a parent. The queue is sorted by requests' deadlines, and only requests whose deadlines are not yet passed remain in the queue. Therefore, the size of the queue is usually very small. If the recently completed request is at the head of an established stream, we let the disk wait for a future request and in the meantime activate a timer for the completed request. Note that the algorithm does not remember every member of a stream. Instead, it only needs to keep track of the most recent request of a stream as well as its current length. When a new request arrives, we examine requests in the *parent-to-be* queue to see if it can extend a stream. If a request in the queue reaches its deadline without seeing a new request as its child, the stream led by the request is usually abandoned. One exception is that when stream has been sufficiently long—when its size is larger than *stream_threshold* by a factor of *tolerance_factor*, or 50% by default—we give the stream a second chance to get extended. When the disk has kept serving a stream for more than a threshold time period (*stream_time_slice*), the disk will dispatch a selected pending request, instead of waiting for a future child request in the stream (not shown in the pseudo code). In our work, we leave the issue of fairness to the external scheduler that has process information, or to the local work-conserving scheduler, such as the Deadline scheduler. When Deadline boosts the priority for dispatching of requests that have waited for too long the stream algorithm respects the decision by immediately sending them to the disk.

```
/* Procedure invoked upon completion of request R*/
R.completion_time = current_time;
R.position = LBN of data requested by R;

/* 'selected_pending_request' is the request selected
   by the work-conserving algorithm */
R.service_time =calculate_service_time(R.position,
                 selected_pending_request.position);
R.deadline = R.completion_time + R.service_time;

/* insert R into the queue sorted by requests'
   deadlines */
queue_of_parent_to_be <-- R;

/* If the stream is established, wait for a
   potential child request */
if (R.stream_size >= stream_threshold) {
   R.timer.timeout = R.service_time;
   activate R.timer;
} else
   dispatch selected_pending_request;

/* Procedure invoked upon arrival of request new_R*/
new_R.arrival_time = new_R's arrival time;
new_R.position = LBN of data requested by new_R;

for each request R in 'queue_of_parent_to_be' {
   if (R.deadline < current_time)
     remove R out of the queue;
   if (new_R.arrival_time-R.completion_time+
   calculate_service_time(R.position, new_R.position)
   < R.service_time) {
     /* new_R is R's child */
     if (R.stream_size >= stream_threshold) {
       turn off R's timer;
       dispatch new_R;
     }
     new_R.stream_size = R.stream_size + 1;
     remove R from queue_of_parent_to_be;
   }
   else
     new_R.stream_size = 1;
}

/* Procedure invoked upon expiration of
   request R's timer */
if (R.stream_size >=
           (1+tolerance_factor)*stream_threshold){
  R.timer.timeout = R.service_time*tolerance_factor;
  R.service_time *= (1+tolerance_factor);
  R.deadline = R.completion_time + R.service_time;
  R.stream_size = stream_threshold;
  activate R.timer;
}
else
  remove R out of 'queue_of_parent_to_be';
```

Figure 2: Stream scheduling Algorithm. In the pseudo code, function *calculate_service_time(disk_pos, req_pos)* is used to calculate the service time when the disk head is at *disk_pos* and the requested data is at *req_pos*, all in terms of LBNs. While we remember only the most recent member request of a stream and the size of a stream, we treat the size as an attribute of the request, denoted as *R.stream_size*.

The forming of streams and scheduling of requests are two independent procedures. That is, no matter what the scheduling decision is, the stream's development is not affected. The forming of streams is determined by the arrival and location of future requests, which usually do not depend on whether the disk actually waits for a child request, though the time period between a request's arrival and its completion is determined by the scheduling decision. Therefore, the stream scheduling algorithm can be used with any work-conserving scheduler. In addition, as the size of the *parent-to-be* queue is small, the algorithm is of low cost, specifically $O(N)$, where N is the size of the queue.

2.2 The Stream Scheduling Algorithm in a Disk Array

The effectiveness of non-work-conserving scheduling algorithms depends on the existence of locality in the requests of a process or a stream. This locality can be sufficiently strong to form an established stream when it is presented to the entire storage system. However, when the storage system consists of an array of disks where data are striped, each disk only sees a subset of the requests and the locality presented to individual disks can be much weaker. As each disk has to be individually scheduled to accommodate its specific data layout and request pattern, instead of all disks being fully synchronized and using one request scheduler [19, 8], it would be hard for each scheduler, on its own, to take advantage of the potential benefit of non-work-conserving scheduling. As an example, for a sequence of synchronous requests $[R_0, R_1, ..., R_{n-1}]$, which could be a stream if they were all served by a single disk, let us assume that only requests R_i (*i mod m* = k) reach disk k, where m is the number of disks in the array ($k = 0, 1, ..., m-1$). After serving R_0, disk 0 would not see R_m until $R_1, R_2, ...,$ and R_{m-1} have been served by other disks, whose service times depend on their respective scheduling decisions and could be significant if long-distance seeks are involved. Even worse, when one request has to access data spread on multiple disks, it is not completed until the last piece of the data is served, and the request's service time can be long if the disks are not coordinated to serve it quickly.

The time period between completion of a request and arrival of the next request of a stream observed at *one* particular disk (such as completion of R_0 and arrival of R_m at disk 0 in the example) consists of two types of time components. One is thinktime, or the time period from the completion of one request to the arrival of the next one of the stream observed by the disk array (such as completion of R_0 and arrival of R_1 in the example stream); another is response time, or the time period from the arrival to the completion of a request in the stream. A request's response time consists of its wait time and service time. To enable non-work-conserving scheduling, we need to minimize the time period for a disk to see its potential child request. While the involved thinktimes cannot be reduced for synchronous requests, the response time can be reduced by dedicating all disks to serving requests of a stream during a certain time period through disk coordination.

As we do not have process information, we set up a disk-array scheduler that treats the disk array as one big virtual disk and uses the method described in the stream scheduling algorithm to identify streams. The disk-array scheduler uses the array's logical addresses for calculating service times and uses pending requests on respective physical disks to evaluate the inequality for identifying child requests. The stream threshold for established streams is increased by m times, where m is the number of disks. Once a stream is established in the virtual disk, which we call a virtual stream, we attempt to find a stream on each physical disk corresponding to the virtual stream, which we call physical stream. Without dedicating all disks to the virtual stream, there is little chance for a physical disk to see its corresponding physical stream because of high response times. However, forcing all disks to serve only the virtual stream's requests before knowing whether the physical streams can be formed runs the risk of idling multiple disks for an excessively long time.

To address the challenge, we do not use a request's actual arrival time to determine whether it can extend a physical stream at a physical disk, as this time might be significantly reduced if all disks were dedicated to the corresponding virtual stream. Instead, we use the arrival time less the response times between the completed request and the disk's next request in the virtual stream (such as the arrival time of R_m minus the sum of response times of requests R_i ($1 \leq k \leq m-1$) in the example stream). The physical streams formed in this way represent the most optimistic estimates on future requests' arrival times, because the response times cannot be reduced to zero even if all disks are dedicated to the virtual stream. Once the array scheduler finds that physical streams have been established on all the disks for a particular virtual stream, it marks the virtual stream's next request to each disk as urgent so that it can be dispatched immediately to bring each disk head to the corresponding physical stream. After this, the array's scheduler instructs each disk's scheduler to use their respective physical stream for non-work-conserving scheduling and use the actual request arrival time to extend the stream. In this way, the non-work-conserving scheduling is certain to be cost-effective even though the physical streams are initiated with optimistic estimates of request arrival times. When a disk's physical stream is broken because it fails to find its next child request, this phenomenon usually cascades to other disks as it would cause other disks' streams to take longer time to see their respective next requests. When the array's scheduler observes broken physical streams, it will mark the virtual stream as *unusable*. Note the scheduler will keep maintaining the virtual stream to prevent a new stream from being formed and triggering non-work-conserving scheduling on the disks once again, which has been shown not to be cost effective. For the disk array, instead of letting each disk decide how long it continuously serves a physical stream, we let the array scheduler determine the time period during which each disk is supposed to serve its physical stream corresponding to the virtual stream. In this way the serving of requests in a virtual stream is fully coordinated across the disks.

3 Performance Evaluation

To evaluate the performance of the stream scheduling framework, we implemented it in the Linux kernel 2.6.31.3, either as a wrapper of a work-conserving disk scheduler to create a stream scheduler for individual disks, or as a revised implementation of the Linux software RAID *mdadm* for a disk array. In the experiments the CPU is an Intel Core2 Duo with 2GB DRAM memory and the disks are 7200RPM, 500GB Western Digital Caviar Blue SATA II (WD5000AAKS) with a 16MB built-in cache. The disk array has five disks connected to the host via a RAID card (RocketRAID 2320).

3.1 Disk Schedulers in Linux

Currently there are four configurable disk scheduler modules in the Linux distributions, each implementing a commonly used scheduler: Noop, Deadline, AS (or Anticipatory), and CFQ. Among them, Noop and Deadline are work-conserving while the other two are non-work-conserving. Noop simply dispatches a request as soon as it is received and does nothing beyond merging contiguous requests. Though it does not sound meaningful when the scheduler is used for dispatching requests directly to the hard disk, it is actually the preferred choice in other cases, such as in guest VMs of virtual machines and the systems using the SAN block device. This not only saves CPU cycles but also allows the requests to reach the lower level as early as possible, where a scheduler can see requests from different guest VMs or hosts and know how data are actually laid out on the disk(s) [32]. For this reason, we include Noop in the evaluation. Deadline is a scheduler approximating CSCAN augmented with a deadline-enforcement mechanism to prevent starvation. AS is a deadline scheduler enhanced with the anticipatory capability to wait for a future request that is of strong locality and is issued by the same process. CFQ aims to fairly distribute disk time among I/O-intensive processes and to bound request response time as Deadline does. As CFQ allows the disk to be idle waiting for future requests, it is non-work-conserving.

3.2 The Stream Scheduling in Linux

In the implementation we place Deadline in the stream scheduling framework and turn it into a non-work-conserving scheduler, the stream scheduler (SS). To accommodate the starvation avoidance mechanism, the stream scheduling algorithm respects the decision made by Deadline about immediate dispatching of expired requests by suspending its dedicated service to a stream. In the evaluation we set *stream_threshold* to be 4. We set *stream_time_slice* to 124ms if not stated otherwise, that is, a stream can be uninterruptedly served for at most 124ms if there are other pending requests in the system. This setting is consistent with that in AS for continuous requests from one process. We will present results of a sensitivity study on the parameter in Section 3.6.

Today's hard disks store multiple requests pending in it and enables its own scheduler such as NCQ for internal scheduling. The disk will continue serving requests pending in it after it completes a request. This poses a challenge to the implementation of the stream scheduling framework because the location of the most recently completed request is not necessarily the disk head position when the request it will dispatch next gets served. For example, when SS decides to idle the disk to wait for a future request by suspending dispatching requests, it assumes that the disk head will stay where it is. However, in a hard disk with stored pending requests, the disk head may have sought to another pending request scheduled by NCQ. To address the issue, we make a customization of the SS algorithm. In the kernel, there is a FIFO queue (*struct request_queue*), into which the disk scheduler dispatches its requests and from which the disk driver takes requests to the disk hardware. In other words, the actual service order will be basically consistent to the order in which the requests stay in the queue, assuming NCQ does not make a major change in the order. Accordingly, the disk head position when the next request is dispatched can be best indicated by the request at the queue tail, or the most recently inserted request. For this reason, SS makes a scheduling decision for the tail request when it is added into the queue, or considers it as the completed request in the stream scheduling algorithm, instead of for the actually completed request. If the decision is to wait for a future request, none of the currently pending requests are allowed to get into the queue and the corresponding timer will be activated at this time. In this way, the assumption made by SS about the disk head location still holds.

To estimate the service time of a request when the disk head is at *disk_pos* and the request is at *req_pos*, all in terms of LBNs (*calculate_service_time(disk_pos, req_pos)*), we adopted a simple empirical method which has been widely used for its effectiveness [25, 14, 16]. In this method, requests of various distances between two adjacent ones are sent to the disk and corresponding service times are collected. A smooth curve is fit through the measured [distance, time] data points and is used to represent *calculate_service_time()* function. In addition, as CSCAN prefers to serve requests in the forward direction, for the same inter-request distance we increase the cost of backward access by 50%.

3.3 Storage without Process Information

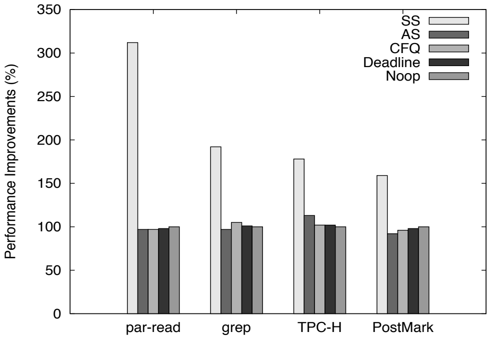


Figure 3: Performance of benchmarks *par-read*, *grep*, *PostMark*, and *TPC-H* with different disk schedulers (SS, AS, CFQ, Deadline, and Noop) when the process information of requests is removed from the workloads. The performance is presented as the schedulers' percentage improvement over that of Noop. For *par-read* and *PostMark* the performance is measured with throughputs, which are 16.0MB/s and 815.9KB/s, respectively for Noop. For *grep* and *TPC-H* the performance is measured with execution times, which are 73.5s and 228.2s, respectively, for Noop.

We first evaluate schedulers of storage systems for which process information for requests is not available, such as hardware RAID, SAN, and iSCSI connected storage devices. As the devices usually use proprietary software and their internal disk schedulers are not open-sourced for instrumentation, we hide process context information from the schedulers, or equivalently we make the schedulers believe that all requests are issued by the same process. In this section, we discuss the experimental results for one disk, and leave those for disk arrays to Section 3.5.

The benchmarks we use in this experiment are *par-read*, *grep*, *PostMark*, and *TPC-H*. *par-read* is a microbenchmark we wrote to study the impact of varying thinktime on the schedulers' performance. It creates four independent processes, each reading a 1GB file using 4KB requests in parallel. There is a 50GB gap between each two adjacent files. By default the thinktime between consecutive requests of a process is set to 0. *grep* is a Linux text search program we run to look for a non-existent word in the Linux 2.6.31 source code tree so that the entire directory tree is read. In the experiment we run two *grep*s, each reading one of two copies of the Linux directory with a 50GB gap between them. *PostMark* is to measure the performance of an Internet server running e-mail, netnews, or e-commerce applications, where random access of small files is the dominant access pattern [26]. In the experiment, we run four PostMark benchmarks (version 1.5.1), each creating a data set consisting of 10,000 files whose sizes are in the range between 0.5KB and 10KB. Each data set is 50GB away from the next data set. *TPC-H* is a decision support benchmark that processes business-oriented queries against a database system to examine large volumes of data. In our experiment we use PostgreSQL 8.3.7 as the database server and use DBT3 1.5.0 to create tables in it. We choose the scale factor 1 to generate the database and run query 19 against it. We run three *TPC-H* instances, with a 50GB space gap between adjacent data sets. Figure 3 shows the performance improvements of the four schedulers (SS, AS, CFQ, and Deadline) over Noop for the four benchmarks.

The experiments demonstrate that without process information both AS and CFQ lose the performance advantages they had enjoyed when they knew which requests are issued by the same process. Each process in the benchmarks synchronously issues its requests. For benchmarks *grep* and *PostMark*, which issue random requests and generally do not trigger prefetching in the operating system, the disk scheduler can see at most one request from a process at a time. Without seeing a nearby pending request, Deadline would dispatch a remote one and constantly move the disk head between remote data sets. This causes its performance to be as low as Noop. Without knowing which process actually issues a request, AS and CFQ assume all requests are from the same process and serve any pending requests when they see them, even if they are in distant regions. Consequently, they degenerate into work-conserving schedulers such as Deadline. However, if we let the information available to AS and CFQ in the experiments, they would perform as well as SS (with a performance difference less than 3%), demonstrating the importance of non-work-conserving scheduling.

Interestingly, the observations for random access can also be made on the other two benchmarks issuing sequential requests, which triggers prefetching in the operating system and allows the scheduler to see asynchronously issued requests. The condition for a work-conserving scheduler to keep serving one process's requests is to eliminate quiet periods in the process's I/O service, or the time period during which the scheduler does not see any requests from the process since last time when the scheduler attempts to

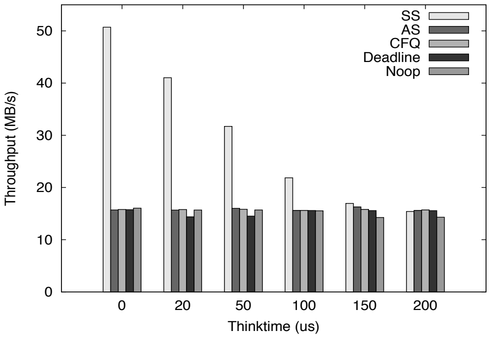


Figure 4: Throughputs of *par-read* with varying thinktimes, the time period between two continuous requests issued by a process.

dispatch this process's request. However, prefetching does not eliminate quiet periods in the system for two reasons. First, Linux maintains two readahead windows to prefetch file data. Prefetch requests issued for one window are contiguous and sent to the scheduler together. The scheduler has a good chance to merge them into one request. Consequently, the next prefetch request would not be triggered and sent to the scheduler until this request is completed and its data is consumed by the process. Second, as today's hard disks store multiple pending requests, a scheduling decision may have to be made before the process's request is completed. At this moment, it is likely the process's next prefetch request has not been generated, creating a quiet period. In both cases, Deadline, as well as AS and CFQ when process information is unavailable, would schedule other process's request and thrash the disk head among processes. While increasing the prefetch window can reduce number of quiet periods, they are unlikely to be fully removed. While SS does not rely on process information, its performance advantage is impressive with about 3.2X throughput improvement over the other schedulers. If we increase the thinktime, the performance improvement of SS becomes increasingly small as their wait times become larger (shown in Figure 4). When the thinktime is as large as $200\mu s$, the corresponding quiet periods increase to as large as about $8.5ms$, which causes streams to break and accordingly causes SS to stop waiting for future requests and behave like Deadline.

3.4 Storage with Inadequate Process Information

Next we consider four benchmarks running in an environment where the process information is inadequate or misleading. To investigate how synchronization of I/O-intensive threads affects behaviors of disk schedulers, we wrote a microbenchmark called *multi-threads*, in which there are four processes, each forking two threads. Each thread reads a 40MB file in a strided pattern, reading the first 4KB of data of every 16KB segment from the beginning to the end of the file. The distance between the two files accessed by one process is 100MB, and the distance of files read by adjacent processes is 50GB. Two threads of a process synchronizes after each makes every five requests. The performance improvements of the schedulers for the benchmark over that of Noop are presented in Figure 5. We can see that SS more than doubles the performance of Deadline in terms of reduction of execution time. Unfortunately AS and CFQ deliver performance even worse than that of Noop. The reason is that the synchronization disrupts their non-work-conserving scheduling, which is unnecessarily tied to the process. For example, assuming that two threads of a process are T_A and T_B, AS keeps serving requests from T_A by anticipatory wait until T_A reaches a synchronization point. Then AS has to wait for about $4ms$ until its timer expires and then it starts to serves T_B's requests, even though a T_B's request is pending nearby. In Linux a thread is presented as a light-weight process. Because the nearby pending request belongs to another thread, AS does not immediately dispatch it. Instead it suffers a long and unfruitful wait. In comparison, without relying on the process information SS is not constrained by the synchronization and dispatches any nearby requests.

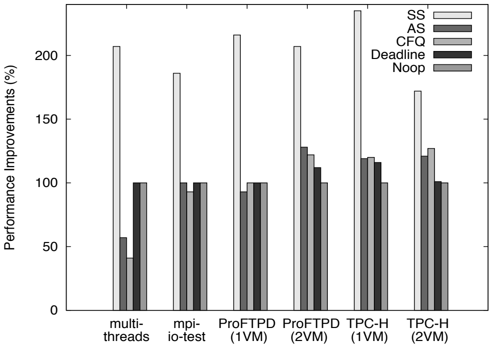


Figure 5: Performance of benchmarks *multi-threads*, *PVFS*, *ProFTPD*, and *TPC-H* with different disk schedulers. *ProFTPD* and *TPC-H* run either on one virtual machine or on two virtual machines. The performance is presented as the schedulers' percentage improvement over that of Noop. For *multi-threads*, *TPC-H(1VM)*, and *TPC-H(2VM)* the performance is measured with execution times, which are 65.7s, 231.4s, and 332.0s, respectively, with Noop. The performance of *PVFS*, *ProFTPD(1VM)*, and *ProFTPD(2VM)*, is measured with throughputs, which are 132.0MB/s, 17.1MB/s, and 12.5MB/s, respectively, with Noop.

PVFS is a parallel file system widely used in high-performance computing clusters [9]. We run the *mpi-io-test* program, an MPI-IO benchmark from the PVFS2 software package [30], on PVFS 2.8.2. The cluster has four compute nodes and eight data servers, where files are striped with a 64KB striping unit. Each data servers has a SATA disk (Seagate Barracuda 7200.10) with NCQ enabled. We run four such programs, each reading a distinct file with 10GB space in between. Each program has eight MPI processes, two per compute node, to read or write one 10GB file. The processes take turns reading 64KB blocks of data sequentially from a 1GB file. For a particular data server, while requests from the same program have strong locality and SS can exploit the locality and achieve an improvement of aggregate throughput for all four MPI programs by 87% over Deadline or Noop, AS and CFQ seriously underperform (Figure 5). On each PVFS server there is a daemon called *pvfs2-server* accepting requests from compute nodes. To achieve asynchrony in its service, the daemon maintains a pool of threads and uses any available thread to dispatch its requests to the kernel. Consequently, AS or CFQ see requests associated with essentially randomly assigned thread numbers and can hardly recognize the locality within requests from the same thread, which leads to disk head thrashing among blocks of different files.

Xen is a virtual machine monitor that allows multiple guest virtual machines (VMs) to run on it [3]. In Xen, guest VMs send requests to their respective virtual block devices, which use the *blktap* mechanism to pass the requests to the kernel driver in the host VM, a privileged virtual machine that does the actual dispatch of I/O requests to disk. In the experiment we run two benchmarks, ProFTPD 1.3.1 and TPC-H, on Xen 4.0.1-rc6 to evaluate the disk scheduler in the host VM while leaving the schedulers in the guest VMs as Noop to quickly release requests into the host VM. ProFTPD is an FTP server [28]. In the test, we run a ProFTPD instance on each guest VM to serve four clients simultaneously downloading four 300MB files, respectively. There are 20GB space gaps between the files. For TPC-H, we use the same experimental setting for each guest VM as described in Section 3.3. From the experimental results shown in Figure 5 we see that SS significantly improves throughput, while AS and CFQ exhibit only limited, if any, improvements over Deadline and Noop because of their lack of process information about requests issued by processes on the same guest VM. When we run two guest VMs, each of the same setting as that in the one-VM scenario, AS and CFQ produce higher throughput improvement as they can differentiate requests from different guest VMs and thus reduce long-distance seeks among data requested by different VMs. Accordingly the relative performance advantage of SS is reduced.

3.5 Storage with Disk Array

To evaluate the performance impact of disk schedulers on the disk array, we select three benchmarks: *par-read*, *TPC-H*, and *PostMark*, whose settings are the same as described in Section 3.3, except that all files are striped over five disks with a 64KB striping unit. The disk array is organized as RAID0. We have also experimented with RAID5 and obtained consistent results. To focus on the performance challenges imposed by data striping on the disk array, we do not hide process information in the test. The experimental results are presented in Figure 6, which shows that for benchmarks of sequential access pattern, such as (*par-read* and *TPC-H*), SS achieves impressive improvements, 114% and 174% over that of Noop, respectively. Without opportunistic synchronization of the disks, the improvements made by AS or CFQ are limited. For example, AS reduces the execution time of *TPC-H* by only 25% while it can reduce the time by 72% when only one disk is used over that of Deadline (see the measurement in Figure 3 for SS, which produces about the same execution time as AS with known process information). The throughput of *par-read* with SS (361MB/s) approaches the peak throughput of the RAID card (around 400MB/s). The sequential access pattern with the help of aggressive prefetching in the RAID is turned into streams on each physical disk in SS, which helps eliminate disk thrashing. However, with the random access pattern of *PostMark*, SS shows minimal improvement as physical streams can hardly be formed.

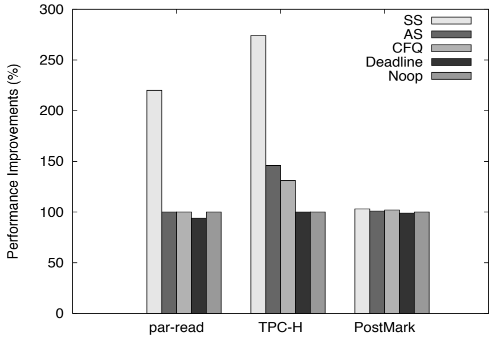
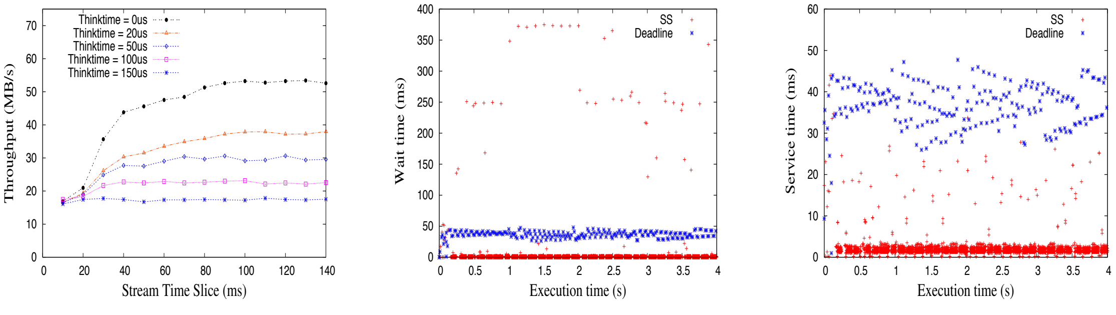


Figure 6: Performance of benchmarks *par-read*, *TPC-H*, and *PostMark*, with different disk schedulers in a 5-disk array. The performance is presented as the schedulers' percentage improvement over that of Noop. Performance of *TPC-H* is measured with execution time, which is 104.6s with Noop. For *par-read* and *PostMark*, it is measured with throughputs, which are 168.0MB/s and 1.3MB/s, respectively, with Noop

3.6 Impact of Stream Scheduling on Throughput and Response Time

SS achieves its performance advantage mostly through its dedication of disk service to one stream of requests during a certain period of time (*stream_time_slice*). By doing so, potentially long distance disk seeks take place only between time slices. Therefore, increasing the time slice is expected to reduce long-distance seeks and thus improve



(a) (b) (c)

Figure 7: Impact of streaming scheduling on throughput improvement and variation of request response time. (a) Throughputs with varying stream time slices for benchmark *par-read* of different thinktimes. (b) Request wait times with SS of default time slice (124ms) for *par-read* of 0 thinktime with SS and Deadline. (c) Request service times with SS of default time slice (124ms) for *par-read* of 0 thinktime for SS and Deadline.

I/O throughput. However, requests that are pending but do not belong to the currently served stream may experience a longer pending period with increased time slice, which can increase variation in response time. To study the effect of the time slice on throughput and response time, we run benchmark *par-read* with an experimental setting the same as described in Section 3.3. As shown in Figure 7(a), the throughput improves with the increasing time slice. The more I/O intensive (with a smaller thinktime) the program is, the larger the improvement. The throughput improves quickly with I/O-intensive programs before the time slice reaches 100ms. After that, further increasing the slice yields only diminishing returns. This is why SS uses the default time slice of 124ms, the same value as adopted by Linux's AS. With this time slice, we measure two components of every request's response time, namely wait time and service time, during the execution of *par-read* with zero thinktime, and show them for the first four seconds of execution with SS and Deadline in Figure 7(b) and Figure 7(c), respectively. Unsurprisingly, SS produces some substantially large wait times (as large as 0.37s), as it rotates its service among four streams with a 124ms slice. Considering that Deadline's default timeout period for boosting request priority is 0.5s, these wait times are deemed acceptable. Meanwhile, as each cycle of such rotation produces only a few long wait times for synchronous requests, the percentage of requests with long wait times is very small and most requests have significantly reduced wait times with SS (Figure 7(b)). Furthermore, the use of a modest time slice in SS, which increases variation of response time, is paid off with significantly reduced request service time (Figure 7(c)) and improved disk efficiency.

4 Related Work

The effectiveness of disk scheduling is highly dependent on the existence of request locality. For this reason, there are many efforts to improve disk access locality. In the high-performance computing field many optimizations are made in the middleware to transform a large number of small non-contiguous requests into a smaller number of larger contiguous requests, including Data sieving [34], Datatype I/O [6], and Collective I/O [34, 43]. Because locality is about requested data locations on disk, there are many efforts to rearrange on-disk data layout to improve spatial locality, including data relocation [15] or data replication, either within one disk [14, 4, 20] or across multiple disks [42]. In addition, compiler techniques can be employed to improve locality by forming preferable I/O access patterns for the disks as well as optimizing file layouts matching known access patterns [18, 21]. However, the enhanced locality can be weakened or even lost when there are multiple processes, each concurrently issuing synchronous I/O requests. The locality can be recovered by non-work-conserving disk schedulers, such as the Anticipatory Scheduler [16]. Anticipatory scheduling has been implemented in some popular Linux disk schedulers including anticipatory [24] and (CFQ) [1].

The problem with the assumption by existing non-work-conserving schedulers on the availability of process information has been recognized in the literature, but effective solutions have not yet been proposed. One scenario is that the disk scheduler in the virtual machine monitor, such as AS, does not know from which specific process running on a guest virtual machine a request is issued. The Antfarm facility can help infer process information for disk scheduling by tracking activities of OS processes [17]. However, application of the technique is limited in the virtual machine environment. In addition, effort must be expended to implement the facility for each individual virtual machine system and the system must be open for instrumentation and patching. The difficulty caused by the lack of process information has also been found with the AS scheduler deployed in the NFS server [11], where the proposed

proach is to use other access context information, such
accessed files' directory or owner, as hints to group re-
uests for scheduling. While this approach can make up for
e inadequacy to some extent, the hints may not be always
levant in revealing on-disk locality to the scheduler and
ould be misleading. A study of the Linux disk schedulers
ound that AS or CFQ can underperform significantly even
hen process information is available but multiple pro-
sses cooperatively send synchronous requests, because
S or CFQ may fail to find anticipation opportunity when
attempts to attribute history access statistics to individ-
l processes [36]. By identifying access streams for non-
ork-conserving scheduling directly from the access loca-
ons, SS discards the requirement for process information
stead of looking for its possibly inadequate substitutes
ith additional overhead in the OS or file systems.

The use of an I/O stream, or request sequence, to ana-
ze and exploit access locality has been used before. Re-
arding I/O prefetching, though many sophisticated de-
gns have been proposed, such as those based on proba-
lity graph model [38], information-theoretic Lempel-Ziv
gorithm [7], or time series model [37], the stream-based
proach dominates the design of prefetching in the system
d has proven its effectiveness and efficiency [27, 41, 35].
reams are also formed on the hard disk addresses to track
sk access history and enable on-disk prefecthing [12].
nother interesting work is a tool called C-Miner that uses
data mining technique to find streams of disk block ac-
ss representing repeatable block sequences, which can
used for initiating reliable prefetching [22]. While SS
so tries to form streams among requests to the disk, the
reams serve a different purpose. For prefetching, a well-
tablished stream will lead to prefetching of multiple data
ocks ahead of stream, while for SS the stream is main-
ined to determine whether the disk should wait for an
coming request. More importantly, the cost of using
reams in the aforementioned works can be much higher
an that for SS when stream members have to be remem-
red for evaluation of stream quality, while SS needs only
track the latest member of a stream.

Regarding scheduling in the disk array, the necessity of
ordinating requests has been widely recognized, espe-
ally for those with small striping units. When multiple
sks are involved to serve a request, *"disks take differ-
t amounts of time to position, the request must wait for
e slowest-positioning disk to transfer its data"* [10]. A
ssible solution is a synchronized interleaved disk sys-
m that synchronizes disk spindles and serves one request
a time in a disk array [19, 8]. However, for striping
it size larger than one byte or for a number of disks in
disk system beyond a certain limit, a fully synchronized
sk array could seriously hurt performance by limiting the
mber of concurrently served requests [31]. The inter-
rence among requests from different processes caused
by uncoordinated disk access has been reported and addressed in the cluster-based storage environment by using a timeslice-based co-scheduling method [40]. Though their work is similar to ours in the coordination of some or all disks and dedication of them to one process at a time, it cannot be effectively used as a disk scheduler to exploit spatial locality for higher performance. One reason is that their work requires an offline-calculated scheduling plan according to QoS specifications that does not adapt to the workload dynamics. Another reason is that it does not evaluate the benefits of dedicated service to a process relative to the cost of disk synchronization, and indiscriminately applies the synchronization to all programs. In contrast, SS dynamically evaluates the cost effectiveness of non-work-conserving scheduling by tracking and validating streams and opportunistically allows the disks to serve one virtual stream at a time. A scheme using opportunistic synchronization to reduce I/O interference among multiple MPI programs accessing a cluster of data servers has been proposed [44]. Without identifying streams, the scheme must assume a file is accessed by only one program and the MPI library and parallel file system must be instrumented to infer the assumed relationship and make it available to the scheduler. In contrast, SS provides a more general solution not constrained by availability of process information.

5 Conclusions

We have described the design and implementation of a stream scheduling framework that turns any work-conserving disk scheduler into a non-work-conserving one, even without process information available, to exploit locality embedded in the sequences of synchronous requests. The framework can also opportunistically coordinate the services at different disks of a disk array to recover and exploit the locality weakened by file striping. The framework has been prototyped in the Linux kernel, both as a disk scheduler and as a software RAID scheduler. Extensive experiments have demonstrated that SS can significantly improve the performance of representative benchmarks such as by TPC-H, PostMark, grep, FTP, as well as MPI programs. In particular, SS shows its unique value in environments where process information is unavailable, such as block or file storage servers and virtual machines.

6 Acknowledgements

We thank Kei Davis, Xuechen Zhang, our shepherd Elizabeth Varki, and the anonymous reviewers for their constructive comments that helped us to improve the paper. This research is supported by U.S. NSF CAREER award CCF-0845711.

References

[1] J. Axboe, "Completely Fair Queueing (CFQ) Scheduler," *http://en.wikipedia.org/wiki/CFQ*, 2010.

[2] D. Boutcher and A. Chandra, "Does Virtualization Make Disk Scheduling Pass?," *ACM SIGOPS Operating Systems Review*, Vol. 44, Issue 1, 2010.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," *Proc. of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[4] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis, "BORG: Block-reORGanization for Self-optimizing Storage Systems," *Proc. of the 7th USENIX Conferenece on File and Storage Technologies*, 2009.

[5] A. Ching, A. Choudhary, K. Coloma, and W. Liao, "Non-contiguous I/O Accesses Through MPI-IO," *Proc. of IEEE International Symposium on Cluster, Cloud, and Grid Computing*, 2003.

[6] A. Ching, A. Choudhary, W. Liao, R. Ross, and W. Gropp, "Efficient Structured Data Access in Parallel File System," *Proc. of IEEE International Conference on Cluster Computing*, 2003.

[7] K. M. Curewitz, P. Krishnan, and J. S. Vitter, "Practical Prefetching via Data Compression," *ACM SIGMOD Record Archive*, Vol. 22, Issue 2, 1993.

[8] 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*, Vol. 26, No. 2, 1994.

[9] P. Carns, W. Ligon III, R. Ross, and R. Thakur, "PVFS: A Parallel File System For Linux Clusters", *Proc. of the 4th Annual Linux Showcase and Conference*, 2000.

[10] P. M. Chen and D. A. Patterson, "Maximizing Performance in a Striped Disk Array,“ *Proc. of 17th annual international symposium on Computer Architecture*, 1990.

[11] H. Chen, J. Xiong, and N. Sun, "A Novel Hint-based I/O Mechanism for Centralized File Server of Cluster," *Proc. of IEEE International Conference on Cluster Computing*, 2008.

[12] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang, "DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch," *Proc. of USENIX Annual Technical Conference*, 2007.

[13] G. Peng and T. Chiueh, "Availability and Fairness Support for Storage QoS Guarantee," *Proc. of IEEE International Conference on Distributed Computing Systems Conference*, 2008.

[14] H. Huang, W. Hung, and K. Shin, "FS2: Dynamic Data Replication in Free Disk Space for Improving Disk Performance and Energy Consumption", *Proc. of the 20th ACM Symposium on Operating Systems Principles*, 2005.

[15] W. Hsu, A. Smith, and H. Young, "The Automatic Improvement of Locality in Storage Systems," *ACM Transactions on Computer Systems*, Vol. 23, Issue 4, 2005.

[16] S. Iyer and P. Druschel, "Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O," *Proc. of the 18th ACM Symposium on Operating Systems Principles*, 2001.

[17] S. T. Jones, A. C. Arpaci-dusseau, and R. H. Arpaci-dusseau, "Antfarm: Tracking Processes in a Virtual Machine Environment," *Proc. of the USENIX Annual Technical Conference*, 2006.

[18] M. Kandemir and A. Choudhary, "Compiler-Directed I/O Optimization," *Proc. of the 16th International Symposium on Parallel and Distributed Processing*, 2002.

[19] M.Y. Kim, "Synchronized Disk Interleaving," *IEEE Transactions on Computers*, Vol. C-35, No. 11, 1986.

[20] R. Koller and R. Rangaswami, "I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance," *Proc. of the 8th USENIX Conferenece on File and Storage Technologies*, 2010.

[21] M. Kandemir, S. Son, and M. Karakoy, "Improving I/O Performance of Applications through Compiler-Directed Code Restructuring," *Proc. of 6th USENIX Conference on File and Storage Technologies*, 2008.

[22] Z. Li, Z. Chen, S. Srinivasan, and Y. Zhou, "C-Miner: Mining Block Correlations in Storage Systems," *Proc. of 3rd USENIX Conference on File and Storage Technologies*, 2004.

[23] E. K. Lee and R. H. Katz, "An Analytic Performance Model of Disk Arrays and its Applications", *Tech. Rep. UCB/CSD 91/660, Univ. of California, Berkeley, Calif.*

[24] A. Morton, "Linux: Anticipatory I/O Scheduler", *http://kerneltrap.org/node/567*

[25] F. Popovici, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Robust, Portable I/O Scheduling with the Disk Mimic," *Proc. of the 2003 USENIX Annual Technical Conference*, 2003.

[26] "The PostMark Benchmark", *www.freshports.org/benchmarks/postmark/*, 2010.

[27] R. Pai, B. Pulavarty, and M. Cao, "Linux 2.6 Performance Improvement through Readahead Optimization", *Proc. of the Linux Symposium*, 2004.

[28] "The ProFTPD Project", *http://www.proftpd.org/*, 2010.

[29] A. E. Papathanasiou and M. L. Scott, "Aggressive Prefetching: An Idea Whose Time Has Come," *Proc. of the 10th Workshop on Hot Topics in Operating Systems*, 2005.

[30] PVFS, http://www.pvfs.org/. Online-document, 2010.

[31] A. L. N. Reddy and P. Banerjee, "An Evaluation of Multiple-disk I/O Systems," *IEEE Transactions on Computers*, Vol. 38, No.12, 1989.

[32] Red Hat, Inc., "Oracle 10g Server on Red Hat Enterprise Linux 5 Deployment Recommendations," *http://www.redhat.com/*, 2008.

[33] E. Rosti, E. Smirni, G. Serazzi, Giuseppe, and L. Dowdy, "Analysis of Non-Work-Conserving Processor Partitioning Policies," *Proc. of the Workshop on Job Scheduling Strategies for Parallel Processing*, 1995.

[34] R. Thakur, W. Gropp, and E. Lusk, “Data Sieving and Collective I/O in ROMIO,” *Proc. of the 7th Symposium on the Frontiers of Massively Parallel Computation*, 1999.

[35] A. J. Smith, “Sequentiality and Prefetching in Database Systems,” *ACM Transactions on Database Systems*, Vol. 3, No. 3, 1978.

[36] S. Seelam, R. Romero, P. Teller, and B. Buros, “Enhancements to Linux I/O Scheduling,” *Proc. of the Linux Symposium*, 2005.

[37] N. Tran and D. A. Reed, “Automatic ARIMA Time Series Modeling for Adaptive I/O Prefetching,” *IEEE Transactions on Parallel and Distributed Systems*, Vol. 15, Issue 4, 2004.

[38] V. Vellanki and A. Chervenak, “A Cost-Benefit Scheme for High Performance Predictive Prefetching,” *Proc. of the ACM/IEEE conference on Supercomputing*, 1999.

[39] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger, “Argon: Performance Insulation for Shared Storage Servers,” *Proc. of the 6th USENIX Conference on File and Storage Technologies* , 2007.

[40] M. Wachs and G. Ganger,“Co-scheduling of disk head time in cluster-based storage,” *Proc. of 28th International Symposium on Reliable Distributed Systems*, 2009.

[41] F Wu, H. Xi, and C. Xu, “On the Design of a New Linux Readahead Framework,” *ACM SIGOPS Operating System Review*, Vol. 42, No. 5, 2008.

[42] X. Zhang and S. Jiang, “InterferenceRemoval: Removing Interference of Disk Access for MPI Programs through Data Replication,” *Proc. of International Conference on Supercomputing*, 2010.

[43] X. Zhang, S. Jiang, and K. Davis, “Making Resonance a Common Case: A High-Performance Implementation of Collective I/O on Parallel File System,” *Proc. of IEEE International Parallel and Distributed Processing Symposium*, 2009.

[44] X. Zhang, K. Davis, and S. Jiang,“IOrchestrator: Improving the Performance of Multi-node I/O Systems via Inter-Server Coordination,” *Proc. of Supercomputing*, 2010.

Improving throughput for small disk requests with proximal I/O

Jiri Schindler, Sandip Shete, Keith A. Smith
NetApp, Inc.

Abstract

This paper introduces proximal I/O, a new technique for improving random disk I/O performance in file systems. The key enabling technology for proximal I/O is the ability of disk drives to retire multiple I/Os, spread across dozens of tracks, in a single revolution. Compared to traditional update-in-place or write-anywhere file systems, this technique can provide a nearly seven-fold improvement in random I/O performance while maintaining (near) sequential on-disk layout. This paper quantifies proximal I/O performance and proposes a simple data layout engine that uses a flash memory-based write cache to aggregate random updates until they have sufficient density to exploit proximal I/O. The results show that with cache of just 1% of the overall disk-based storage capacity, it is possible to service 5.3 user I/O requests per revolution for random updates workload. On an aged file system, the layout can sustain serial read bandwidth within 3% of the best case. Despite using flash memory, the overall system cost is just one third of that of a system with the requisite number of spindles to achieve the equivalent number of random I/O operations.

1 Introduction

This paper focuses on an important but neglected aspect of file system performance: workloads that mix random writes with sequential reads to the same data. In particular, serial reads after random writes (SRARW) are common in many applications that are large consumers of storage in enterprise environments. For example, database systems typically acquire and update data through online transactional processing (OLTP), which is dominated by small writes, and subsequently read it in bulk for other tasks, such as analysis or backup. SRARW workloads are particularly problematic in large-scale deployments, which are often spindle-limited and too large to be moved to flash-based SSDs cost effectively.

Existing file system designs optimize either the serial read access or the random writes in a SRARW workload, but do so at the expense of the other operation type. At one end of the spectrum, *write-anywhere* file systems such as the Sprite log-structured file system (LFS) [27] and its descendants [19, 22, 3] are write optimized. They batch small or random writes into larger sequential allocations on disk, transforming updates of logically unrelated data into physically sequential I/O. However as they age, their data layout becomes fragmented, scattering related data across the disk. Thus, logically sequential access such as a database table scan leads to inefficient disk I/O. We have measured access to physically fragmented data at as little as 3% of the best-case serial read bandwidth. (See results in Section 5.3.)

At the other end of the spectrum, *update-in-place* file systems, such as FFS [21] and related designs [5, 32] are optimized for serial read and write access. These file systems attempt to allocate logically sequential data to physically sequential disk locations, providing good bandwidth for serial data access. However, this translates into inefficient non-sequential I/O, as destination blocks are predetermined by past allocation decisions, which are unlikely to be optimal in the face of random updates. Moreover, when such systems keep older versions of the data, they must perform a variant of copy-on-write [25], doubling the amount of inefficient random write I/O. Database systems often decouple this inefficient back-end I/O from foreground processing through the use of logging. The log then becomes a staging area for asynchronous bulk updates to the database tables. However, this technique alone does not mitigate the high cost of random I/Os to the back-end of a storage system that has limited I/O capacity.

To increase the back-end's effective I/O capacity without increasing disk (spindle) count, we introduce a new type of disk access pattern that we term *proximal I/O*. We demonstrate how proximal I/O leverages features of current disk drives to retire in a single revolution several I/Os scattered across dozens of tracks holding hundreds of thousands of sectors (Section 2). We propose a new data layout (Section 3), which shares the desired properties of existing copy-on-write, write-anywhere file systems that make random user writes efficient and allow for snapshots with minimal I/O overhead.

Using a prototype implementation of our data layout (detailed in Section 4), we show that with write cache sized only at 1% of the overall storage capacity, we can service 5.3 I/Os per revolution for workloads with random updates, all the while maintaining data layout on a heavily aged system that can deliver 97% of the bandwidth achieved with the best-case scenario of physically sequential layout (Section 5). The 5.3 I/Os serviced per revolution represent a 7× improvement over the cost of a random disk I/O in traditional frame arrays with update-in-place data layout. With RAID5 and non-volatile write caches, they use four random disk I/Os at the back-end; with copy-on-write snapshots, this number doubles. In contrast, our approach uses about one disk I/O at the back end for every user-level random update when *both* RAID and copy-on-write are employed.

The primary contribution of this paper is the introduction of a data layout strategy that combines a small amount of flash memory with proximal I/O to efficiently service random updates to sequentially-allocated on-disk data without undermining on-disk locality. We provide efficiency both in performance and cost, significantly improving the performance of random writes at less than a third of the cost per IOPS of a pure disk solution. Finally, we present the first study to quantify the behavior of modern disk drives under proximal I/O access pattern.

2 Proximal I/O

Proximal I/O leverages the ability of modern disk drives to execute multiple I/Os per single revolution scattered across dozens of disk tracks. Given the 300-400 Gb/in^2 aerial density of the magnetic media in currently shipping disk drives, this translates to a range of hundreds of thousands of logical blocks (*LBN*s)[1]. We describe the interplay between seek-time profile and request scheduling that make proximal I/O possible.

In the material presented here, we focus primarily on one disk model (the Seagate Cheetah 15K.5, introduced to the market in late 2007). However, both the 15,000 RPM enterprise-class drives as well as the 7,200 RPM high-capacity nearline drives, colloquially referred to by their interfaces as respectively SCSI/FC and SATA drives, are capable of proximal I/O. Our measurements of over 20 different models of both drive types, representing several generations of the same family of drives and manufactured by four different vendors confirm the observations about proximal I/O described here.

2.1 Relevant technologies

Historically the seek profile, the plot of seek time as a function of radial distance, has been described by a continuous curve with two components: for small distances, one that is a square root of the cylinder distance and, for larger seek distances, one where seek time is a linear function of cylinder distance [28]. As observed by Schlosser et al. [31], the seek profile of more recent disks includes a third component: for very small seek distances of less than *C* cylinders, the seek time is nearly constant and is effectively equivalent to the track-switch time, or the time needed for the head to settle on a track.

The surface-serpentine disk layout adopted by more recent drives [1, 31] uses minimal seek time over a range of tracks. After mapping the last *LBN* to a given track, the disk firmware maps the next *LBN* to the adjacent track on the same surface rather than to the same track on a different surface. After mapping across *C* consecutive tracks, the next logical *LBN* is mapped to a sector on a track of a different surface *C* cylinders away. Thus, when accessing sequential run of *LBN*s, the disk heads will occasionally seek across the *C* cylinders to access the next logically sequential *LBN*. Using a disk extraction tool [29], we determined $C = 65$, which covers the range of 624,000 *LBN*s (1200 SPT × 65 cylinders ×8 surfaces) for the 300 GB Cheetah 15K.5 disk.

The Shortest-Positioning-Time-First (SPTF) request scheduler [34] implemented in the disk firmware can effectively increase the throughput of serviced requests. It reorders requests to minimize the total positioning time (i.e., the sum of seek time and rotational latency) for each I/O request in the queue. With sufficiently large number of outstanding requests, it can lower the total positioning cost (i.e., the sum of the seek time and rotational latency) and service many more requests per unit of time [1].

2.2 Expressing request service time

Issuing only one request at a time to the disk negates the benefits of the SPTF scheduler. The service time of a small random request will then be equivalent to the sum of average seek (equivalent to 1/3 of the full-strobe seek) and rotational delay of 1/2 a revolution. For the Cheetah 15K.5 disk, this is is respectively 3.6 ms and 2 ms, resulting in service time of 5.6 ms. For a 7,200 RPM Western Digital RE3 nearline disk, the values are respectively 6.9 ms and 4.2 ms, yielding service time of 11.1 ms.

As these times for the two drives vary (by design) by a factor of 2×, we will use instead a relative measure of service time, here called *OP*, that lets us ignore the differences between disk types and their generations. Thus, 1 *OP* is the service time for a small random disk request or the measure of resources consumed when servicing a random disk I/O.

Our enterprise-class disk has the capacity to service about 180 I/Os per second, while the nearline disk only 90. These drives demonstrate a useful rule of thumb;

[1] The number of sectors per track (SPT) for recent 3.5” disk drives ranges between 800 and 2800. edge. The 15,000 RPM enterprise-class disks employing 65 mm platters have fewer SPT at their outer-most track compared to the 95 mm platters in the 7,200 RPM disk drives [2].

the average seek time of a disk is roughly equal to the time for a full rotation. Thus, the seek component of an OP is roughly 2/3 *OP* while the remaining portion is attributed to the rotational latency of half a revolution. This rule predicts that a disks can typically service 0.66 random requests per revolution. Our two drives do slightly better—0.71 for the enterprise-class disk and 0.75 for the nearline disk. This trend has held across many disk generations with different rotational speeds and seek times.

2.3 Measurements

To quantify the benefits of proximal I/O, we measured the per-request service time in a batch of requests, here called a strand, under a variety of conditions. We chose at random a location on the disk and controlled the span of *LBN*s covered by the requests as well as the number of requests in the strand issued simultaneously (i.e., the disk queue depth). The measured response time for the entire strand, listed in Table 1, is the sum of the service times of the individual requests in the strand.

To service a strand of requests, the disk must first seek to the general vicinity of the requests. Servicing the first request in the strand thus incurs the cost of an average seek in addition to some rotational latency. However, if the requests are near each other, servicing the remaining requests, incurs only some additional rotational latency and potentially minimal seek/track switch, since all requests in the strand are within *C* cylinders of each other. As we batch all requests, the disk is free to reorder them.

Figure 1 shows the effective per-request service time as the number of requests in the strand (and hence the queue depth) increases, expressed both absolutely and in relative units of *OP*s. The graph compares three different access modes. The track-approximate access limits the *LBN* span to 1024, the approximate size of the disk's track, rounded down to the power of two. The proximal access uses a span of 100,000 *LBN*s. The semi-sequential access represents the best possible disk access after sequential streaming [30] — the requests are carefully chosen such that each request is positioned at a different track and at an offset equivalent to the minimal seek/track switch time. For semi-sequential access, we need to know detailed disk drive parameters. On the other hand, proximal I/O does not require the knowledge of track switch time or precise track size (SPT).

	Strand response time (ms)			
Requests per strand	2	4	6	8
Semi-sequential	5.9	7.3	8.6	9.9
Track-approximate	7.4	10.8	13.9	17.0
Proximal READ	7.4	11.2	14.2	17.1
Proximal WRITE	8.6	12.9	16.8	20.4

Table 1: Comparison of strand response times for Seagate Cheetah 15K.5. The mean service time of single READ request is 5.6 ms. For WRITE, it is 5.8 ms due to extra write-settle time.

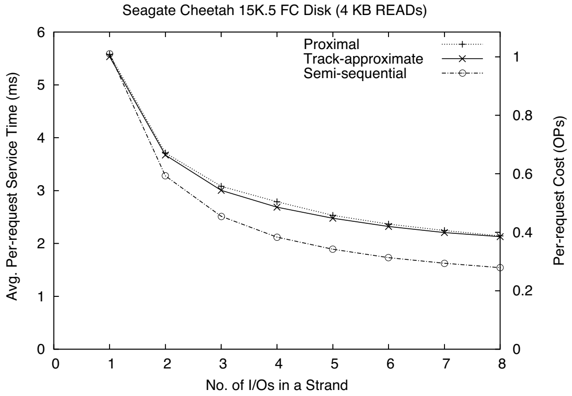


Figure 1: Per-request cost of small requests in a strand.

We remark on the following trends. First, as the number of requests in the strand increases, the effective per-request service time decreases from 1 *OP* to 0.39 *OP* for a strand of 8 requests — a 2.5× improvement over the case with one request per strand. Second, both track-approximate and proximal mode are very similar, despite the ten-fold difference in the *LBN* span. And third, the semi-sequential mode yields 0.23 *OP* for 8 requests per strand compared to 0.39 *OP* of proximal mode — an additional 1.7 times improvement. The results for WRITEs (not shown here) exhibit a similar trend; the slightly higher strand response time (see Table 1) is due to additional write-settle time for track-to-track seeks.

2.4 Detailed model comparisons

Two hypotheses might explain why we do not see values for proximal I/O that are closer to the semi-sequential mode. First, with randomly chosen blocks, some of them may land on the same track and the disk firmware opts to prefetch the remainder of the track before servicing other requests. Second, even without triggering prefetching, the random placement of the requests can cause extra (missed) revolutions as we describe below.

The semi-sequential access carefully chooses the placement of blocks so as to eliminate any rotational delays between requests after a track switch. With randomly chosen requests in proximal access, requests on different tracks can have rotational offset that is smaller than the time needed to switch tracks. The following paragraphs help illustrate how the disk scheduler minimizes overall rotational latency. They also show that it is the stochastic nature of the request placement rather than an artifact of the disk firmware causing the extra revolutions.

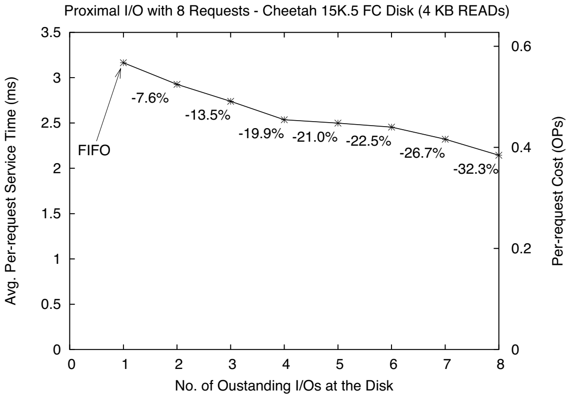


Figure 2: Per-request service time for strand of 8 requests with a varying number of READ requests outstanding at the drive. The first point is equivalent to a FIFO scheduler. The percentages next to the data points represent improvement relative to the FIFO data point. The second graph compares the modeled and the observed (measured) behavior.

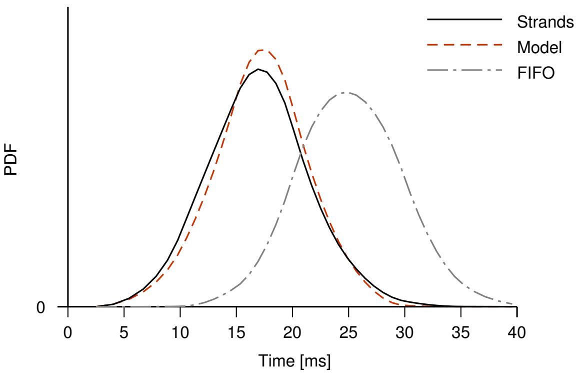


Figure 3: Model-predicted vs. measured (observed) values.

To verify our hypothesis, we set up an experiment, whose results are shown in Figure 2, where we fixed the number of requests per strand to 8 and varied the number of requests queued at the drive. With one request outstanding, the requests are serviced in FIFO order. As the number of outstanding requests increases, the disk scheduler can choose a request with smaller rotational latency, yielding a 32% reduction in per-request service time for a queue depth of 8 requests. This result confirms that proximal access effectively leverages the SPTF scheduler.

To obtain the expected number of revolutions needed to service a strand of requests with proximal I/O, we developed an analytical model for computing the expected strand response time and the probability distribution of missing revolution(s) for proximal I/O access. The model is based on the birthday paradox principle [33] and works at a high level as follows. It divides the disk into equally sized wedges or bins. When two requests (on different tracks) fall into the same bin, the disk heads cannot move fast enough to reach the second request in time and will have to service it during the next revolution. Because of the high track switch time relative to the revolution time (0.4–0.8 ms), there are only a few bins (days in a month) available and several requests are likely to fall into the same bin (i.e., having birthday on the same day). See Appendix A for model details.

Figure 3 demonstrates the high accuracy of our model, comparing the measured and modeled distributions of the strand response times. The two curves labeled Strands and Model are very similar with nearly identical distributions. The curve labeled FIFO corresponds to measurements with one request outstanding at the disk drive, which is the scenario described in Figure 2.

2.5 Practical considerations

There are a few practical considerations for proximal I/O. First, we discovered that the manner in which we issue the requests in the strand is important. Issuing requests in a random order and relying solely on the scheduler's ability to reorder them does not work. However, when we issue the requests in ascending *LBN*, the scheduler works as expected — it picks from the strand a request with the lowest positioning cost, services it first, and reorders the remaining ones as necessary. Figure 4 shows the effect of issuing requests in ascending *LBN* for two different disk drives. The previously reported results include this workaround.

We attribute this limitation to two factors: the lack of the embedded CPU power (especially for nearline drives) and a firmware bug. We consulted disk manufactures who acknowledged both factors. In one case, our inquiry led to a fix in a subsequent firmware release. In practice, even with current limitations, pre-sorting requests is not an issue. Second, to engage the request scheduler properly, the strand must be issued to a non-empty disk queue. Again, in practice this limitation is not a problem. In many deployed systems, disks are seldom idle; they are busy servicing either client-generated workload or a variety of background scanning and grooming tasks. Third, we explored strands with at most 8 requests outstanding, although deeper queues would likely result in better results. This is again driven by a practical consideration. Many commercial storage systems [20, 8, 9] limit the number of pending requests to 4 or 8 to put a bound on the response time of a time-critical request.

As a final remark note that our experiments assumed a purely random workload, which we simulated by a uniform distribution of requests in the given range of *LBN*s. We believe that workloads that have more locality (but which are not sequential by nature) will benefit at least as much (if not more) from using proximal I/O.

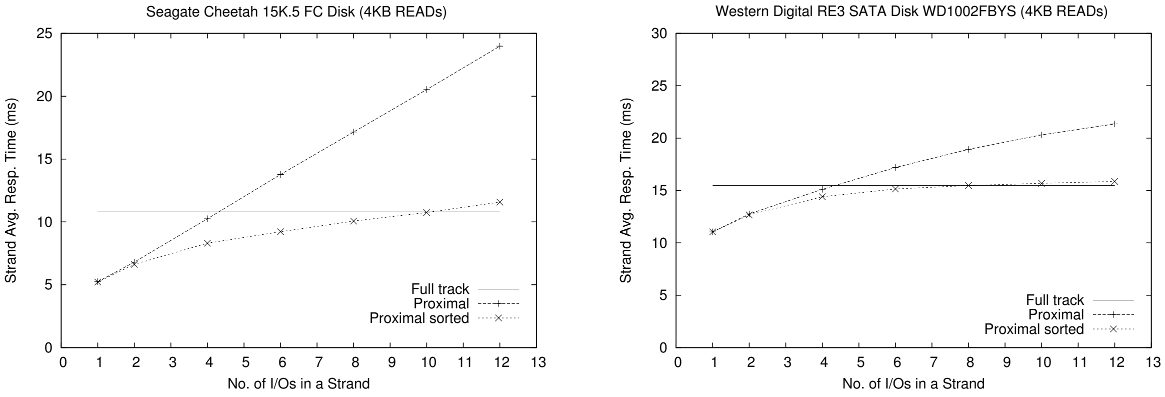


Figure 4: The effects of sorting requests in ascending *LBN* order before issuing them to the disk. The full track data serves as a reference of request scheduling efficiency and corresponds to reading approximately one full track. Same trends hold for WRITEs.

3 Data layout with proximal I/O

Previous section explained how proximal I/O can retire several I/Os per revolution within the span of approximately 100 tracks or half a million *LBN*s. In this section we describe the design principles for data layouts that leverage the proximal I/O construct. For our discussion, we will use the term block to designate the basic data allocation unit in file systems (typically 4 KB) to distinguish them from sectors, or *LBN*s, which are typically 512 bytes and serve as the basic unit of disk I/O.

3.1 Increasing I/O density

We start by considering how to increase the density of writes in the SRARW workload. The goal is to take a stream of random requests and produce sequences of I/O that will benefit from proximal I/O. As long as a storage system can produce a batch of several, say eight, write requests, and the data layout engine can place them within the span of $\approx 100{,}000$ blocks, each request will be serviced in time equivalent to much less than one revolution and consume only 3.2 *OP* of resources (i.e., 0.4 *OP* per request as shown in Figure 1) regardless of the previous position of the disk heads. In contrast, servicing eight blocks randomly scattered across the entire media will require 8 *OP*. Put differently, we need to find a way to increase the effective I/O density instead of spreading out a given batch of I/Os across the entire disk.

We use two complementary approaches to achieving the necessary I/O density. First, we leverage indirection when assigning data to their physical locations akin to inodes in file systems that map file offset to a physical address at the underlying storage. Write-anywhere systems with no-overwrite semantics [19, 27] already take advantage of this approach; random writes at the storage system interface are mapped to the same segment (allocation area) at the physical layer. Our technique expands on this notion by allocating data to free space in the vicinity of the previously written logically related data. Second, when the I/O density of 6–8 requests within the zone of effectiveness of proximal I/O is not enough, we complement the new type of write allocation described above with the use of a staging area. With a large-enough staging area, one can selectively pick appropriate requests and write-allocate them to achieve the required I/O density as determined by the disk technology.

Our approach contrasts with existing ones in several ways. Traditional frame arrays that export logical volumes composed of disk drives organized into a RAID group typically do not have much flexibility in mapping their blocks to the underlying devices. They stripe data in a round-robin fashion across the constituent disks. Such systems do not require any additional metadata; they can compute the disk number and disk offset with simple modulo and divide arithmetic. However, a given write operation at the storage interface will land at a specific location on the disk, negating the desire for decoupling the front-end workload from the back-end. As a result, they need much larger write caches to achieve the required write density compared to our approach.

A back-end system with hundreds of large-capacity near-line disk drives, will require hundreds of GB of staging area. Using that much NV-RAM (i.e., some form of battery-backed DRAM) would make the overall system cost prohibitively high (although there are commercial systems that offer such configurations [10, 9]). A more cost-effective solution is to use Flash memory an append-only log [23], at approximately 1/10th of the cost per GB. Another possibility is to use a dedicated disk as commercial relational database systems do for their log. However, with disk-based staging area, we would require some additional DRAM to hold the data during the destage operation to perform random reads from DRAM rather than disk during destage operation.

3.2 Overwrites and snapshots

In contrast to purpose-built storage systems [4, 7, 14, 18]) that function as fixed-content repositories or that handle specialized scientific workloads that write lots of intermediate results, most writes in commercial systems are (logical) overwrites or updates rather than new writes. For example, commercial databases and email servers [11] update individual records within database pages. New, or append-only, writes occur infrequently as these systems typically pre-allocate their table space by writing out empty (but non-zero) pages in bulk. Similarly, writes to objects containing virtual machine disk images create clones of a baseline golden image with relatively few unique blocks.

Proximal I/O can also reduce the overhead of preserving multiple versions of the same block, be it snapshots for fast recovery after a crash or keeping diverging replicas of original files. A storage system will turn an update into a copy-on-write operation that will write data to a new location. A write-anywhere file layout, for example, lends itself to keeping snapshots with very little overhead, as in WAFL [19]. Other systems, such as frame arrays, with direct mapping of logical blocks to disk locations must issue an extra I/O to preserve old block versions.

Both types of systems exhibit a similar shortcoming. A version (the new one in case of WAFL and the old one in case of frame arrays) of the data is put in a location that is convenient for the system *without* considering the semantic relationship to the original data. This can adversely affect the efficiency of subsequent reads. Logically related data may end up too far away from each other, incurring high positioning cost when they are both first written and then later on retrieved. Therefore, when a data layout engine maintains physical proximity of logically related data (be it a live version or a snapshot), it can leverage proximal I/O for copy-on-write of data that are updates in place from the client's perspective.

Most storage systems use RAID to protect their data against disk failures and grown media defects. The RAID read-modify-write (RMW) operation is not proximal I/O per se. However, we can combine copy-out and RMW operations and leverage proximal I/O; we can pipeline them such that we write out the just-read old version of the data within the effective span of proximal I/O in time before the disk spins around to write the new version of the block. With enough flexibility in the data layout, we can accomplish two RMW operations, that is four media accesses, in time equivalent to slightly more than one and a half revolution plus the initial seek.

3.3 Efficiency of reads

So far we have discussed proximal I/O in the context of writes. However, it can also improve the efficiency of subsequent reads. The careful placement of related versions of the data during writes allows the disk to collect physically non-contiguous blocks with minimal positioning overhead for logically sequential access. Proximal I/O can access both the current data as well as the snapshots with similar efficiency.

Systems that do not place logically related data near each other are likely to perform differently depending on which version of the data they access. For example, a sequential table scan on an aged system may be less efficient than one performed against a snapshot made earlier. In contrast, when systems can place related data within the span of blocks that can be serviced with proximal I/O, they will likely exhibit much smaller variations in performance regardless of the version/snapshot they are reading. This is because both the old and new versions of data blocks, as well as logically related unmodified blocks will be in close proximity of each other, allowing proximal I/O to read either old or new versions of the data with high efficiency.

3.4 Summary of key design points

We summarize the key design points of a data layout engine suited for proximal I/O:

1. Flexible mapping of object data to the physical on-disk location is an effective mechanism for increasing I/O density. Put differently, given a certain level of "randomness" in the front-end workload, systems with flexible per-block location pointers will need smaller staging area compared to systems that use rigid mapping of front-end blocks to on-disk physical locations.

2. The system needs to employ large-enough write staging area to achieve the required I/O density for the given front-end workload. Naturally, completely random workloads will require the largest size. In practice, workloads are rarely purely random — there are typically hot spots where relatively small portion of the data is updated frequently. These hotspots reduce the amount of staging area required for effective proximal I/O.

3. A data layout engine with built-in efficient copy-on-write mechanism is well suited for proximal I/O; only some adjustments will be necessary to marry the constraints of proximal I/O with their already existing mechanisms.

We conclude by examining the various access patterns encountered by enterprise storage systems. In addition to serial reads after random updates (SRARW) that we target with proximal I/O, we must consider random reads, sequential writes, and sequential reads not coupled with frequent (small) updates. It is our belief that storage systems already employ effective techniques that can handle these other access patterns. In our view, proximal I/O is the missing link that fixes the inefficiencies of disk accesses in today's deployments.

Increasing the size of read caches, for example by employing devices based on flash memory, effectively copes with random reads. A modicum of NV-RAM can turn sequential writes into efficient disk accesses. In contrast, increasing the efficiency of random updates requires large buffers. Finally, sequential reads (in the absence of small updates) are easy to achieve. For example, workloads typical for fixed-content repositories often write out and read entire objects. When individual objects would be too small for efficient disk I/O, they are grouped into larger allocation and access units [4, 35].

3.5 Target workloads

Our work targets workloads in large-scale storage systems and is motivated by the emergence of virtualized data center environments. The computing infrastructure includes a storage manager that allocates data from the underlying storage systems in large chunks or extents. For example, both the Oracle ASM [12] and VMWare VMFS [13] allocate data in 1 MB chunks. Storage systems in these environment in turn provide data management features such as fine-grain snapshots, writable clones, etc. [20].

In these environments SRARW workloads are typical. The storage manager reads and prefetches data in full allocation units (chunks). However updates are typically at a finer granularity—for Oracle ASM, the update size is equivalent to the DBMS page size (typically 4–8 KB). For VM hypervisors, the update size is governed by the block size of the file system in the VM guest operating system. The writes from the storage managers to the underlying storage systems may turn into copy-on-write (rather than update-in-place) operations in order to preserve older versions data for disaster recovery. Our work focuses on these logical update-in-place operations with serial reads for prefetching or OLAP data scans.

4 Prototype data layout engine

The goal of our work is not to build an entire new file system. Instead, we have built a *data layout engine* (DLE) that uses our staging and allocation algorithms to demonstrate the feasibility of using proximal I/O to greatly improve random write performance while maintaining (near) optimal serial performance for SRARW workloads. We believe these algorithms are readily adaptable to both update-in-place and write-anywhere file systems.

Our DLE is, in effect, a stripped-down object storage system. We store logical extents of data in a flat namespace, where each extent is named only by a unique ID. Extents can be created, read, written (and overwritten) and deleted. For simplicity, we only support reads and writes that are properly-aligned on block boundaries. Our DLE includes all of the necessary file systems structures to support this functionality, inode-like structures for each extent, allocation maps to track free space, and additional metadata to facilitate layouts friendly to proximal I/O.

Because we are primarily interested in addressing the SRARW workload, our DLE is designed to efficiently support moderately large extents (1 MB or larger)—large enough for the serial read portion of the workload to benefit substantially from sequential layout. Our DLE works correctly for smaller extents, but we have not tested or optimized its performance in those cases. We believe that for those workloads, file systems would benefit more from using allocation algorithms that are different from those implemented in our prototype DLE. We describe here only the major pieces of our prototype necessary to understand the experiments presented in Section 5.

4.1 Extent interface

The DLE operates on extents. An extent is a contiguous logical range of bytes. The DLE decides how to best allocate extent data into fixed-size blocks (4 KB in our prototype) of the underlying storage subsystem—a logical volume created from raw disks in a RAID group. Internally, each extent is represented by an inode, which is the root of a constant-height tree of indirect blocks. The leaf nodes of this tree contain the extent data.

4.2 Staging area

Our DLE uses a separate flash device as a staging area to accommodate random writes. When the DLE writes data into the staging area, it also updates the corresponding metadata including inode and indirect blocks for the just-written extent. Thus the staging area is the full-fledged home (albeit temporary) for new data, rather than a write cache with a copy of the data. When the system achieves the required I/O density (or the staging area runs out of capacity) we use proximal I/O to move data from the flash-based staging area to the on-disk location. More importantly, during destage, we make just-in-time allocation decisions for the best on-disk placement in relation to the data already-allocated to the disk. Because of our desire to keep previous versions of data for snapshots, we don't overwrite in place and instead write to a new location. When the destage operation finishes, we update the extent and DLE metadata accordingly to reflect its new on-disk location.

As part of its metadata, the staging area maintains a table mapping each block in the staging area to the extent and offset that the block belongs to. This allows us to efficiently locate items in the staging area for destage. Our DLE also uses the flash device to store its internal metadata, so metadata access does not interfere with the SRARW access patterns we wish to study.

4.3 Allocation policies

The more interesting feature of our prototype is the set of write allocation policies we implemented. When new data is written to an extent, we use the size of the write to determine whether to write the data to the staging area or directly to disk. In the current implementation this threshold is 168 KB—a number chosen to be approximately the break-even point between the response time of a random I/O of that size and a full-track read for our current disks. Although, we have not examined alternate settings, we believe that the precise value has little qualitative effect on our system; it only serves to distinguish between small and large writes.

We have three different I/O allocation scenarios in our system: small writes allocated to the staging area, large writes allocated on disk, and collections of small writes allocated on disk when destaging. We manage the staging area as an append-only log. Other more involved schemes are possible, but we have not explored them. When the staging area fills, we destage its full contents and start refilling it again from the beginning. As described earlier, when we write a block to the staging area, we update the metadata that points to it, freeing any on-disk block containing older data at that offset if that block is not used for a snapshot.

When we receive a large write request, we write it directly to disk, allocating new space if necessary, as when first writing an extent. Since we assume that extents will be large, and we want to provide good serial performance, we map large sequential ranges of an extent to similarly sized physical extents on disk that we call *allocation ranges*. By allocating at first fewer physical blocks than the size of the allocation range, we can provide extra space for future updates and write-anywhere-style snapshots [19] at the cost of a corresponding fraction of serial bandwidth. We have not yet explored this capability in our prototype.

We follow the recommendations of Chen et al. [6] for stripe unit size to approximate the disk track size. Given the current disk parameters, we chose 1 MB as the size for the RAID stripe unit and allocation range for nearline drives and 512 KB for enterprise-class drives. Note however, that we need not know the precise disk parameters. The allocation unit size is a configurable parameter in our allocation algorithm and can loosely follow technology trends over time as track size increases.

Small writes (or updates) are first written to a staging area and held there until sufficient number of random updates is accumulated to achieve the required proximal I/O density. At that point we collect the relevant data (using our metadata info) and destage them to their final place. That is where alternative storage technologies work to our advantage; we can perform random reads directly from the staging area backed by e.g., flash memory. If using disks instead, we need to perform (possibly multiple) sorting pass(es) and use additional DRAM.

Destage is a two phase process. First, by scanning the staging area tables, we identify sets of blocks that can be allocated together. We do this by sorting the blocks first by extent and then by logical offset within each extent. Second, from the extent metadata, we determine the allocation range(s) that contain related data i.e., data at the logical offsets immediately preceding and following the data being destaged. If there is enough space in the corresponding allocation range we simply write-allocate data there. When no additional space exists, we look for another allocation range that has enough free space to absorb the blocks and is in the vicinity of proximal I/O.

In the worst case we inspect up to approximately 100 allocation ranges (given current disk characteristics) for each group of blocks i.e., all blocks in the staging area belonging to a single extent and that are logically offset by the range of proximal I/O. In practice, this number is much smaller; when we wrote blocks to the staging area, we typically deallocate the older version of the block on disk, unless they are kept for snapshots. If we are destaging to an allocation range that had no underlying physical storage (i.e., we are writing to a sparse extent), we first allocate a physical extent for the allocation range, and then allocate the destage blocks within it. Figure 5 illustrates the destaging process, showing the layout of data in both flash memory and disk.

Our allocation algorithm uses two parameters dependent on disk technology trends: (a) the SPT governs the efficient allocation and serial I/O size and (b) minimal seek time governs the effective range of proximal I/O. The first parameter dictates the size of our allocation range, the second one, expressed in the number of allocation ranges, provides the flexibility in our allocation decisions during destage operation. The parameters need not closely follow the technology trends. One adjustment for every few disk generations is sufficient. The trends are evolving to our advantage (see discussion in Section 5.4).

4.4 RAID Layer

Our user-level prototype also includes a stand-alone RAID subsystem, which presents a logical volume abstraction to our DLE. This has several benefits compared to using an off-the-shelf one such as hardware RAID controller or a software implementation such as the `md` block device driver in Linux (our prototype platform).

Our RAID implementation offers fine control over scheduling requests to individual disks. We use Linux SCSI generic device (`/dev/sg`) interface that bypasses the kernel block device's buffer cache and the block device schedulers. Linux can issue SCSI commands directly to both SCSI/FC and SATA drives thanks to the libsata layer. Most importantly, our own implementation more closely emulates the operation of an enterprise-class RAID layer and includes features that are missing from the aforementioned RAID implementations.

First, we perform updates either by addition or subtraction so as to minimize the number of disks engaged in I/O operations. Second, just like many other RAID subsystems [20, 8, 9], we maintain additional information for every data block, including, a write generation number for lost write protection and additional data checksum. Since SATA disks support only 512-byte sectors, we must use a separate sector for the additional per-block information. We use 64 bytes of additional information per single 4 KB block grouped into one checksum block for every 63 data blocks, emulating the features of Data ONTAP [20] running on systems with commodity SATA disks. Thus, accessing one block above the DLE interface results in two distinct block accesses.

5 Results

We evaluate the effectiveness of proximal I/O using our DLE prototype. We first study random updates to large extents comprising a logical volume (LUN) exported by a storage system. and then analyze serial reads after our volume has been aged with many small random updates.

5.1 Experimental setup

Our prototype runs as a user-level process on a host with one dual core 3 GHz Intel CPU under Linux 2.6.24 (from stock Ubuntu Server 9.04 distribution). We use a 4+1 RAID4 of 1 TB Western Digital RE3 (WD1002FBYS) SATA drives. We chose these 7200 RPM drives despite their lower performance compared to their enterprise-class counterparts because they are more cost effective.

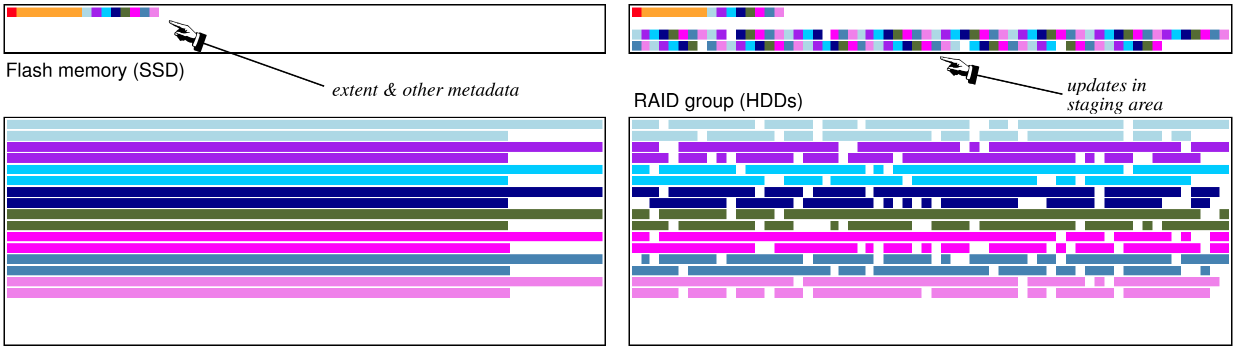
We fill our DLE with 16 MB extents to 89% of its capacity, writing them serially directly to the disks. We then issue 2000 small (4 KB) random updates per extent, thus re-allocating half of all blocks we initially wrote. Our DLE accumulates these updates to an SSD-based staging area, destaging them to back-end disk each time the staging area fills. For measuring serial reads after writes, we read every single extent from our aged DLE (in random order). These requests for 974 logically serial blocks at a time (governed by the fan-out of our indirect blocks) result in several scatter-gather disk I/Os. Figure 5 shows the layout during the execution of updates.

Before the DLE issues a set of requests to the RAID layer, we execute random I/Os to each of the constituent disks so as to avoid "short-stroking" (i.e., generating artificially short seeks due to using only a subset of the disk capacity). We wait for the disks complete the random I/Os and exclude these unrelated I/Os from our analysis. Executing a set of small updates results in many more individual disk I/Os than there are requests in the batch — the RAID layer needs to access the additional checksum blocks and to perform read-modify-write operations.

5.2 Random updates

The results for the random updates are summarized in Table 2. Each table row represents measurements with a different size of the staging area relative to the RAID group size. We collect statistics for each *batch* of I/O, where one batch is the disk I/O generated in destaging the accumulated changes to a single extent. Thus, a entire destage operation will generate one batch of I/O for each extent with at least one block in the staging area. We measure the mean response time and the number of user updates for each batch (columns 2 and 3). These times reflect the disk activity (i.e., the operations on the data). The DLE and extent metadata are updated on the SSDs, on average, with fewer than three I/Os for each batch. Since the SSD I/O service time is much smaller, the disk I/O dominates the batch response time.

We collect the service time for each disk I/O (computed as the difference between the completions of the last two I/Os). We list the mean number of disk I/Os (reads and writes across all five drives) in column 4. Column 5 shows the I/O amplification, the mean number of disk I/Os needed for each user-initiated update. Column 6 shows the equivalent number of disk I/Os serviced per revolution. Finally, we show for the data and parity disks the mean number of I/Os, per-I/O service time, and the resulting disk utilization.



(a) Initial setup with sequentially written extents.

(b) Aged layout with updates still in the Flash memory staging area.

Figure 5: An example of block allocation in the prototype DLE. Initially, extents are written out directly to the RAID group. The Flash memory (SSD) holds extent and DLE metadata. Random updates are first put into the Flash staging area. As the layout ages, the extents are no longer contiguously laid out. However, the DLE maintains the proximity of the related blocks of the same extent by "pluging" holes in the layout created by overwrites two destage operations.

Stage Area	Resp. time	User writes	Disk I/Os	I/O ampl.	I/Os p. rev.	Data disks I/Os	Data disks ST	Data disks Util.	Parity disk I/Os	Parity disk ST	Parity disk Util.
Baseline	16.8 ms	1	8	8x	2.0	4	4.2	97%	4	4.2	97%
1%	129.5	47.5	295.3	6.2x	5.3	43.3	2.0	65%	122.0	1.0	91%
2%	155.0	85.2	465.3	5.5x	7.2	66.1	1.4	61%	201.1	0.7	93%
3%	182.9	119.7	614.8	5.1x	8.1	86.2	1.2	57%	269.9	0.6	94%
4%	227.6	149.3	723.7	4.8x	7.7	99.8	1.1	47%	324.3	0.6	86%
5%	235.9	179.1	847.6	4.7x	8.7	117.0	1.0	48%	379.8	0.6	95%
6%	278.4	226.1	1014.7	4.5x	9.0	136.3	0.9	44%	469.3	0.6	97%
7%	315.4	259.5	1151.5	4.4x	9.0	155.6	0.9	42%	529.2	0.6	97%
8%	320.9	254.8	1166.7	4.6x	8.7	163.5	0.8	43%	512.6	0.6	96%

Table 2: Random updates for various sizes of the staging area. Resp. time is the response time of the batch of user I/Os being destaged and ST is the the per disk I/O service time, both reported in milliseconds. I/O ampl. is the ratio of disk I/Os to user writes. The I/Os per revolution represents the number of I/Os serviced by a drive averages across both data and parity disks. The base data represents a system without staging area, whereby every user write results in RMW at the RAID back-end.

In our baseline data, we show the performance when a batch contains exactly one block. This has a latency of 16.8 ms, and results in 8 separate disk I/Os (an I/O amplification of 8×). Writing a single block to RAID4 results in 4 individual disk I/Os—a read and write of both the data disk and the parity disk. Updating the checksum incurs an additional 4 I/Os.

Next, we explore how our write allocation, coupled with 1% of staging area, leverages proximal I/O to improve the efficiency of disk accesses. We observe that, on average, 47.5 user updates result in 295.3 disk I/Os for a 6.2× amplification that are serviced in 139.5 ms. The per-I/O service time for the data and parity disk is thus 2.0 ms and 1.0 ms respectively. Even though the RAID4 parity disk is more efficient, it has to service many more I/Os and thus is the bottleneck.

Since the batch of I/Os is serviced by proximal I/O, we can retire on average 5.3 I/Os per revolution. Yet, as shown in Figure 1, we were only able to retire 1.9 I/Os per revolution in a strand of 8 requests (17.1 ms to retire 8 read requests with 4 ms rotational time). We achieve this improvement because of the greater I/O density; we are writing strands that contain many more blocks, typically within a single track or two. Also, the RAID layer must also update the checksum block for data blocks that are being written out. This further increases the number of disk I/Os, but also the I/O density — for most data blocks the checksum block is on the same track. For the same reason, we also see only a 6.2× write amplification (instead of 8×); we need to access the same checksum block only once for several data block updates.

As the size of the staging area increases, the batch size increases (from 47.5 to 369.7 updates for staging area of 1% and 10% respectively) and the destage operation for each batch becomes more efficient. The I/O amplification decreases from 6.2× to 4.5× and the number of disk I/Os serviced per revolution grows from 5.3 to 8.6.

5.3 Serial reads after random updates

Table 3 shows the results for sequential reads from an aged layout as depicted in Figure 5(b). Our system reads up to 974 logically consecutive blocks. Given the extent sizes, our DLE requests on average 819.2 blocks that are returned with a mean response time of 9.5 ms. This translates to effective bandwidth of 86.2 MB/s per disk (345 MB / 4 data disks in RAID group). Because of fragmented layout, the request for up to 974 logical blocks results in a batch of 47.5 logically sequential runs of blocks issued to the RAID group that are further broken into individual per-disk I/Os. Because of striping and the need to access the checksum block in addition to the data blocks, each disk services on average 20.7 individual I/O requests. Given the 1 MB stripe unit size on our RAID setup, the original request for 974 logically sequential blocks is typically serviced by three disks.

We repeated the same experiment on the non-aged data layout depicted in Figure 5(a) where extents are laid out on physically contiguous disk sectors. We measured mean response time of 9.2 ms, which translated to per-disk bandwidth of 89 MB/s. Thus, sequential reads after random updates on our system are within 3% of the the best-case scenario of physically contiguous layout.

Finally, we evaluated the performance of serial reads after random updates with write-anywhere-style allocation. In our system, we induced this behavior by eliminating the staging area and writing out data by greedily plugging the holes created by deletes of earlier versions of the data (we did not implement an LFS-style segment cleaner). In this setup, logically serial data increasingly dispersed over the disk over time, resulting in dramatically lower bandwidth compared to the baseline case.

	Per disk statistics			
	Read BW	**Diff**	**I/Os**	**Util.**
Baseline	89.0 MB/s		11.7	85%
Aged layout	86.2 MB/s	-3%	20.7	82%
Write-anywhere	2.6 MB/s	-97%	210.2	85%

Aged layout reads – detailed statistics			
	mean	**min**	**max**
Request response time (ms)	9.5	6.5	32.9
Request size (4 KB blocks)	819.2	200	974
Requests per batch	43.9	28	114
Span of blocks	1002.8	914	1008
Number of I/Os per disk	20.7	2	58
Per-disk resp. time (ms)	8.8	0.9	32.8

Aged layout – read response time quantiles

10%	20%	30%	40%	50%	60%	70%	80%	90%	95%
7.4	7.5	7.6	7.7	7.9	8.1	8.4	22.1	27.8	31.9

Table 3: Serial reads after random updates.

5.4 System cost and technology trends

Lowering overall cost is one of the driving forces behind changing the internal architecture and design of commercial enterprise-scale storage systems. The adjustments to the write allocation policies presented here coupled with deployment of some additional device(s) for the staging area is but one example of such force. Making the prevalent access pattern (e.g., the serial read after random write described here) more efficient allows the system to run workloads with larger I/O demand for the same dollar cost. We now explore the trade off between the cost of additional hardware for the staging area and the resulting improvement in the back-end disk I/O capacity.

Consider the WD1002FBYS disk drive we used in our experiments. It has a measured average seek time of 7.5 ms and rotational speed of 7,200 RPM. With the time of 8.4 ms for a single rotation, the mean time to service a random I/O is 11.7 ms. Thus our drive can perform 86 random IOPS. With a street price of $130, this means we are paying $1.52 per IOPS. Now consider the effects of adding 1% of capacity as a flash staging area. Table 2 shows that in this configuration we can write 5.3 blocks in a single revolution. Adding an average seek means that our system performs 5.3 writes in 15.9 ms, or 3 ms per write. This is equivalent to 333 random IOPS, an improvement of 289% over the basic disk solution.

Adding the flash staging area increases the cost of the system. With cost of flash at $3.13/GB, based on a 160 GB Intel X25 SSD with a street price of $500, a 1% staging area for our 1 TB drive requires 10 GB of flash, increasing our cost by $31.30, to a total of $161.30 for a configuration capable of 333 random IOPS. Thus, in our system, the cost is $0.48 per IOPS, less then a third of the per IOPS cost of the raw disk drive.

With our system, we pay an extra 25% to add a flash staging area and in return we get nearly a 3× performance increase on random writes, while preserving near sequential on-disk layout.[2] Note that these numbers are pessimistic. They assume that the staging area is scaled to the entire disk-based storage capacity. In reality, the staging area need only be 1% of the write working set, further reducing the flash costs.

We conclude by considering the impact of technology trends on the effectiveness of proximal I/O. The disk trends are in our favor. Growing areal media densities

[2] These numbers are shown here only to illustrate our point. our simplified model considers only the cost of individual devices. Also, we ignore many practical system issues such as RAID group size, etc.

(i.e., the increase in both SPT and track density), increase the span of *LBN*s over which proximal I/O will be effective. Larger span gives more options to our system to lay out its data. Similarly, as the flash memory cost decreases, the relative size of the deployed staging area may likely increase relative to the disk storage capacity. This will also increase the effectiveness of our destage process. In the end however, the ratio of flash memory to disk capacity will be driven by customer needs and their ability to get the right performance for the least cost.

6 Related Work

Our work explores the design principles for a data layout suitable for the SRARW workload while leveraging a more efficient disk access pattern. We review some additional related work not mentioned earlier in the paper.

Our data layout is similar in principle to the data journalling mode employed by some journalling file systems [5, 26, 32]. As in those systems, we write data initially to a designated staging area (journal, separate device etc.) and later on destage them to their final location. The difference in our approach, which utilizes proximal I/O, is that the efficiency of our destage operation is much much higher; journalling file systems typically write to a specific location on the disk constrained by their “overwrite-in-place” policy. In contrast, we destage data with fewer constraints offered by the span of blocks in proximal I/O. Additionally, we can consider the best location with respect to the related data and thus make the write operations more efficient.

The Disk Caching Disk (DCD) [24] explored a different technique for using write caching to improve storage system performance. DCD aggregates small writes on a separate caching disk, achieving serial performance when flushing dirty data from the buffer cache. During idle periods it destages data from the cache disk to its home on the primary disk. This design improves the latency of small writes, but does not leverage proximal I/O to achieve better I/O efficiency. A similar technique has also been used in database systems [16].

The idea of proximal I/O combines and expands on the observations about (1) efficient disk access across adjacent tracks with minimal positioning cost [30] and (2) minimal positioning cost when seeking across an ever-increasing range of cylinders [31]. Unlike semi-sequential access, however, proximal I/O does not require detailed knowledge of disk geometry or specialized device interface that provides the position of the next semi-sequential block relative to the current position; it works on systems with standardized interfaces (SATA or SCSI) and off-the-shelf commodity disk drives.

Our DLE design relies on write-anywhere allocation, similar to LFS [27], WAFL [19] and related designs such as ZFS [22] and btrfs [3]. Like these systems, it never overwrites old data in place, making it possible to preserve older versions of data, or snapshots, with minimal I/O overhead. Traditional write-anywhere file systems batch temporally related dirty data for efficient disk writes. Thus logical data locality is lost, requiring segment cleaning [27] or other defragmentation techniques [15] to re-establish sequential layout. In contrast, our DLE allocates data close to logically related on-disk data, preserving logical locality with proximalI/O plus the staging area to achieve efficient write performance.

The Loge [17] disk controller represents another variation of write-anywhere; it virtualizes block addresses so that it can write incoming data at the free locations nearest to the current disk head location. However, the work does not target SRARW workloads; it explicitly assumed that randomly written data would also be randomly read. In principle, many aspects of our DLE design could be implemented in a Loge-like disk controller rather than in a file system, although it would loose the semantic information about which blocks of data are logically related and are likely to be read together. Moreover, our design does not require detailed knowledge of disk head position and thus is time-invariant.

Appendix A: Proximal I/O model

Our objective is to find the expected number of revolutions needed to serve D requests in a strand. Recall that a strand is a collection of proximal I/Os that are sent together to a disk and that are close enough such that servicing any one of the requests incurs a minimal seek equivalent to head/track switch time.

Assume there are SPT sectors per track and D requests, each of size S sectors, and a seek between each request in the strand equivalent to head switch time, H. We express H in terms of the number of sectors that pass by the disk head during track switch. We can formulate the problem of finding the expected number of revolutions in terms of binning requests into B bins. Each request of size S is then randomly placed into any one of the K slots along a circular track. This is analogous to a roulette wheel with K slots and D balls spun simultaneously.

With such a formulation, if two balls (i.e., requests on different tracks) fall into the same bin, that is, if they are within $K \times S/H$ slots, the disk arm cannot service those requests in a single revolution and we get

$$B = \frac{SPT}{H} = \frac{K \times S}{H}.$$

Let's express the probability, Q_1, that no two requests are in the same bin. This is analogous to the probability that no extra revolutions are required when servicing a schedule of D requests in a strand. This can be solved by the birthday paradox problem, where we look for the probability that no two people out of a group of n people in a room have a birthday on the same day out of b possible, and equally likely, birthdays.

$$Q_1 = \frac{b!}{(b-n)!b^n}$$

Using our analogy, we have B bins, which is equivalent to the b possible birthdays, and D, the number of requests in a strand, is the number of people, n. This is equivalent to not using any extra revolutions (since each request is in a separate bin) when servicing the D requests.

We can now calculate the probability that at least one extra revolution will be required as

$$P_1 = 1 - Q_1.$$

More generally, the probability that a birthday is shared by exactly k (and no more) people is expressed as [33]

$$Q_k(n,b) = \sum_{i=1}^{\lfloor n/k \rfloor} \left(\frac{n!b!}{b^{ik} i!(k!)^i (n-ik)!(b-i)!} \sum_{j=1}^{k-1} Q_{k-1} \frac{(b-i)^{n-ik}}{b^{n-ik}} \right)$$

This is equivalent to the probability of servicing a given strand in k revolutions or using exactly $k-1$ extra revolutions. This assumes that each request landed on a separate track and requires a track switch when servicing it.

The probability that we will require at least k extra revolutions in servicing a request (or that $k+1$ or more people share a birthday in our analogy), we have

$$P_k = 1 - \sum_{i=1}^{k} Q_i$$

Now let's express the probability that we will not use any extra revolutions when servicing a strand as a function of number of sectors, H, that pass by during track switch time. With values for the Seagate Cheetah 15K.5 disk's first zone we have $SPT = 1200$, track switch time 0.475 ms, $H = SPT \times \lceil 0.475 \text{ ms}/4 \text{ ms} \rceil = 142$, and the number of bins $B = 8.45$. Therefore, we set $\lfloor B \rfloor = 8$, meaning that this disk can at best schedule 8 proximal I/Os in a revolution when the requests are properly offset from each other. With strand where $D = 8$, the probability of not using any extra revolutions is close to zero.

We express the expected number of revolutions for servicing a strand of D requests as

$$E[\text{Revs}] = \frac{1}{2} + \sum_{i=1}^{D} iQ_i(D, SPT/H)$$

For $D = 8$, we get $E[\text{Revs}] = 3.4$, assuming that each request lands on a separate track. Normalized (or per-request) number of revolutions is then 0.43.

Next, we assume eight requests in a strand even though this disk can service at best six in a single revolution. We choose the value of eight because it gives, on average, 12% lower per-request service time compared to a strand with $D = 6$. Adding an initial average seek of 3.5 ms for each strand, the per-request service time is 2.16 ms or 17.26 ms for the entire strand of $D = 8$ with variance $\sigma^2 = 9$ ms. This comes to within 1% of the measured mean service time of 17.14 ms with $\sigma^2 = 9.6$ ms.

Finally, we examine the probability of using exactly one, two, three, and so on, revolutions when servicing a strand of $D = 8$ requests. From our model, the most prevalent value is two extra revolutions (three in total). When $D = 6$ (with $H \approx 7$ for our disk), the probability of not using any additional revolutions is still only 0.02.

Acknowledgments

We would like to thank the program committee and the anonymous reviewers for their helpful comments. In particular, we thank our shepherd, Arif Merchant, for his insight and suggestions for improving this paper. Thanks also to Steve Kleiman for asking the questions that started us down this path and for many fruitful conversations along the way. Finally, we thank Jeffrey Heller, Art Harkin, and the all of NetApp's Advanced Technology Group (ATG) for their advice, support, and suggestions—and for making it fun to come to the office everyday!

References

[1] Dave Anderson. You don't know jack about disks. *Queue*, 1(4):20–30, 2003.

[2] Dave Anderson, Jim Dykes, and Erik Riedel. More than an interface—SCSI vs. ATA. In *Proceedings of Conference on File and Storage Technologies (FAST)*, pages 245–257, 2003.

[3] Valerie Aurora. A short history of btrfs. `http://lwn.net/Articles/342892`, Jul 2009.

[4] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a needle in haystack: Facebook's photo storage. In *Proceeding of Symposium on Operating Systems Design and Implementation (OSDI)*, Oct 2010.

[5] Mingming Cao, Theodore Y. Tso, Badari Pulavarty, Suparna Bhattacharya, Andreas Dilger, and Alex Tomas. State of the art: Where we are with the ext3 filesystem. In *Proceedings of Ottawa Linux Symposium (OLS)*, pages 69–96, Jul 2005.

[6] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: high-performance, reliable secondary storage. *Computer Surveys*, 26(2):145–185, 1994.

[7] EMC Corporation. EMC Centera: Content Addressed Storage. http://www.emc.com/products/systems/centera.jsp, 2007.

[8] EMC Corporation. CLARiiON CX4 Series. http://www.emc.com/products/series/cx4-series.htm, 2010.

[9] EMC Corporation. Symmetrix DMX-4: Enterprise networked storage system. http://www.emc.com/products/detail/hardware/symmetrix-dmx-4.htm, 2010.

[10] IBM Corporation. IBM System Storage DS8000 Turbo. http://www-03.ibm.com/systems/storage/disk/ds8000/index.html, 2010.

[11] Microsoft Corporation. Microsoft Exchange Server. http://www.microsoft.com/exchange/2010/en/us/default.aspx, 2010.

[12] Oracle Corporation. Oracle database storage administrator's guide 11g release 1 (11.1). http://download.oracle.com/docs/cd/B28359_01/server.111/b31107/toc.htm, 2008.

[13] VMware Corporation. VMware vSphere. http://www.vmware.com/products/vmfs/index.html, 2010.

[14] Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowski, Cristian Ungureanu, and Michal Welnicki. HYDRAstor: a scalable secondary storage. In *Proceedings of Conference on File and Storage Technologies (FAST)*, pages 197–210, 2009.

[15] John K. Edwards, Daniel Ellard, Craig Everhart, Robert Fair, Eric Hamilton, Andy Kahn, Arkady Kanevsky, James Lentini, Ashish Prakash, Keith A. Smith, and Edward Zayas. FlexVol: flexible, efficient file volume virtualization in WAFL. In *Proceedings of the 2008 USENIX Annual Technical Conference*, pages 129–142, Jun 2008.

[16] Klaus Elhardt and Rudolf Bayer. A database cache for high performance and fast restart in database systems. *ACM Transactions on Database Systems*, 9:503–525, 1984.

[17] Robert M. English and Alexander A. Stepanov. Loge: a self-organizing disk controller. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 237–251, Jan 1992.

[18] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. *SIGOPS Operating Systems Review*, 37(5):29–43, 2003.

[19] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Proceedings of USENIX Winter 1994 Technical Conference*, pages 235–246, Jan 1994.

[20] NetApp Inc. Data ONTAP 8. http://www.netapp.com/us/products/platform-os/data-ontap-8/, 2010.

[21] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for unix. *Transactions on Computer Systems*, 2(3):181–197, 1984.

[22] Sun Microsystems. ZFS at OpenSolaris community. http://opensolaris.org/os/community/zfs/.

[23] Sun Microsystems. Logzilla: Hybrid storage pools in the 7410. http://dtrace.org/blogs/ahl/2008/11/10/hybrid-storage-pools-in-the-7410/, 2008.

[24] Tycho Nightingale, Yiming Hu, and Qing Yang. The design and implementation of a DCD device driver for unix. In *Proceedings of USENIX Annual Technical Conference*, pages 295–308, Jun 1999.

[25] Zachary Peterson and Randal Burns. Ext3cow: a time-shifting file system for regulatory compliance. *Transactions on Storage*, 1(2):190–212, 2005.

[26] Hans Reiser. Reiserfs. http://www.namesys.com/.

[27] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10:1–15, 1992.

[28] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27:17–28, 1994.

[29] Jiri Schindler and Gregory R. Ganger. Automated disk drive characterization. Technical Report CMU-CS-99-176, Carnegie Mellon University, 1999.

[30] Jiri Schindler, Steven W. Schlosser, Minglong Shao, Anastassia Ailamaki, and Gregory R. Ganger. Atropos: A disk array volume manager for orchestrated use of disks. In *Proceedings of Conference on File and Storage Technologies (FAST)*, pages 159–172, Mar 2004.

[31] Steven W. Schlosser, Jiri Schindler, Stratos Papadomanolakis, Minglong Shao, Anastassia Ailamaki, Christos Faloutsos, and Gregory R. Ganger. On multidimensional data and modern disks. In *Pro-*

ceedings of Conference on File and Storage Technologies (FAST), pages 225–238, Dec 2005.

[32] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS file system. In *Proceedings of USENIX Annual Technical Conference*, pages 1–14, Jan 1996.

[33] Eric Weisstein. Birthday problem. `http://mathworld.wolfram.com/BirthdayProblem.html`, 2007.

[34] Bruce L. Worthington, Gregory R. Ganger, and Yale N. Patt. Scheduling algorithms for modern disk drives. *SIGMETRICS Perform. Eval. Rev.*, 22(1):241–251, 1994.

[35] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of Conference on File and Storage Technologies (FAST)*, pages 269–282, Feb 2008.

FastScale: Accelerate RAID Scaling by Minimizing Data Migration

Weimin Zheng, Guangyan Zhang
Tsinghua University
gyzh@tsinghua.edu.cn

Abstract

Previous approaches to RAID scaling either require a very large amount of data to be migrated, or cannot tolerate multiple disk additions without resulting in disk imbalance. In this paper, we propose a new approach to RAID-0 scaling called FastScale. First, FastScale *minimizes data migration*, while maintaining a uniform data distribution. With a new and elastic addressing function, it moves only enough data blocks from old disks to fill an appropriate fraction of new disks without migrating data among old disks. Second, FastScale *optimizes data migration* with two techniques: (1) it accesses multiple physically successive blocks via a single I/O, and (2) it records data migration lazily to minimize the number of metadata writes without compromising data consistency. Using several real system disk traces, our experiments show that compared with SLAS, one of the most efficient traditional approaches, FastScale can reduce redistribution time by up to 86.06% with smaller maximum response time of user I/Os. The experiments also illustrate that the performance of the RAID-0 scaled using FastScale is almost identical with that of the round-robin RAID-0.

1 Introduction

Redundant Array of Inexpensive Disks (RAID) [1] was proposed to achieve high performance, large capacity and data reliability, while allowing a RAID volume to be managed as a single device. As user data increase and computing powers enhance, applications often require larger storage capacity and higher I/O performance. To supply needed capacity and/or bandwidth, one solution is to add new disks to a RAID volume. This disk addition is termed *"RAID scaling"*.

To regain uniform data distribution in all disks including the old and the new, RAID scaling requires certain blocks to be moved onto added disks. Furthermore, in today's server environments, many applications (e.g., e-business, scientific computation, and web servers) access data constantly. The cost of downtime is extremely high [2], giving rise to the necessity of online and real-time scaling.

Traditional approaches [3, 4, 5] to RAID scaling are restricted by preserving the round-robin order after adding disks. The addressing algorithm can be expressed as follows for the i^{th} scaling operation:

$$f_i(x) : \begin{cases} d = x \bmod N_i \\ b = x/N_i \end{cases} \quad (1)$$

where block b of disk d is the location of logical block x, and N_i gives the total number of disks. Generally speaking, as far as RAID scaling from m disks to $m+n$ is concerned, only the data blocks in the first stripe are not moved. This indicates that almost 100 percent of data blocks have to be migrated no matter what the numbers of old disks and new disks are. There are some efforts [3, 5] concentrating on optimization of data migration. They improve the performance of RAID scaling by a certain degree, but do not overcome the limitation of large data migration completely.

The most intuitive method to reduce data migration is the semi-RR [6] algorithm. It requires a block movement only if the resulting disk number is one of new disks. The algorithm can be expressed as follows for the i^{th} scaling operation:

$$g_i(x) = \begin{cases} g_{i-1}(x) & \text{if } (x \bmod N_i) < N_{i-1} \\ f_i(x) & \text{otherwise} \end{cases} \quad (2)$$

Semi-RR reduces data migration significantly. Unfortunately, it does not guarantee uniform distribution of data blocks after subsequent scaling operations (see section 2.4). This will deteriorate the initial equally distributed load.

In this paper, we propose a novel approach called *FastScale* to redistribute data for RAID-0 scaling. It accelerates RAID-0 scaling by minimizing data migration.

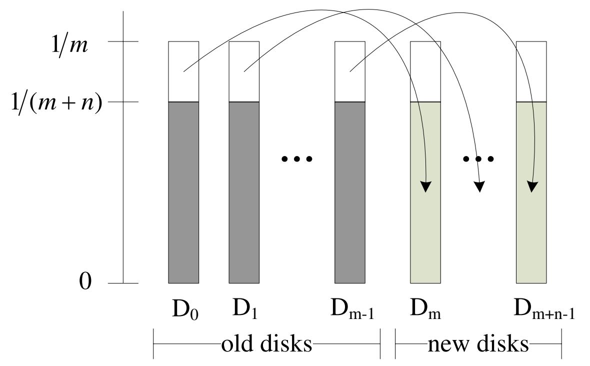


Figure 1: Data migration using FastScale. Only data blocks are moved from old disks to new disks for regaining a uniform distribution, while no data is migrated among old disks.

As shown in Figure 1, FastScale moves only data blocks from old disks to new disks enough for preserving the uniformity of data distribution, while not migrating data among old disks. Consequently, the migration fraction of FastScale reaches the lower bound of the migration fraction, $n/(m+n)$. In other words, FastScale succeeds in minimizing data migration for RAID scaling.

We design an elastic addressing function through which the location of one block can be easily computed without any lookup operation. By using this function, FastScale changes only a fraction of the data layout while preserving the uniformity of data distribution. FastScale has several unique features as follows:

- FastScale maintains a uniform data distribution after RAID scaling.
- FastScale minimizes the amount of data to be migrated entirely.
- FastScale preserves a simple management of data due to deterministic placement.
- FastScale can sustain the above three features after multiple disk additions.

FastScale also exploits special physical properties to optimize online data migration. First, it uses aggregate accesses to improve the efficiency of data migration. Second, it records data migration lazily to minimize the number of metadata updates while ensuring data consistency.

We implement a detailed simulator that uses DiskSim as a worker module to simulate disk accesses. Under several real-system workloads, we evaluate the traditional approach and the FastScale approach. The experimental results demonstrate that:

- Compared with one of the most efficient traditional approaches, FastScale shortens redistribution time by up to 86.06% with smaller maximum response time of user I/Os.
- The performance of the RAID scaled using FastScale is almost identical with that of the round-robin RAID.

In this paper, we only describe our solution for RAID-0, i.e., striping without parity. The solution can also work for RAID-10 and RAID-01. Although we do not handle RAID-4 and RAID-5, we believe that our method provides a good starting point for efficient scaling of RAID-4 and RAID-5 arrays.

2 Minimizing Data Migration

2.1 Problem Statement

For disk addition into a RAID, it is desirable to ensure an even load on all the disks and minimal block movement. Since the location of a block may be changed during a scaling operation, another objective is to quickly compute the current location of a block.

To achieve the above objectives, the following three requirements should be satisfied for RAID scaling:

- Requirement 1 (*Uniform data distribution*): If there are B blocks stored on m disks, the expected number of blocks on each disk is approximately B/m so as to maintain an even load.
- Requirement 2 (*Minimal Data Migration*): During the addition of n disks to a RAID with m disks storing B blocks, the expected number of blocks to be moved is $B \times n/(m+n)$.
- Requirement 3 (*Fast data Addressing*): In a m-disk RAID, the location of a block is computed by an algorithm with low space and time complexity.

2.2 Two Examples of RAID Scaling

Example 1: To understand how the FastScale algorithm works and how it satisfies all of the three requirements, we take RAID scaling from 3 disks to 5 as an example. As shown in Figure 2, one RAID scaling process can be divided into two stages logically: data migration and data filling. In the first stage, a fraction of existing data blocks are migrated to new disks. In the second stage, new data are filled into the RAID continuously. Actually, the two stages, data migration and data filling, can be overlapped in time.

For the RAID scaling, each 5 sequential locations in one disk are grouped into one *segment*. For the 5 disks, 5 segments with the same physical address are grouped

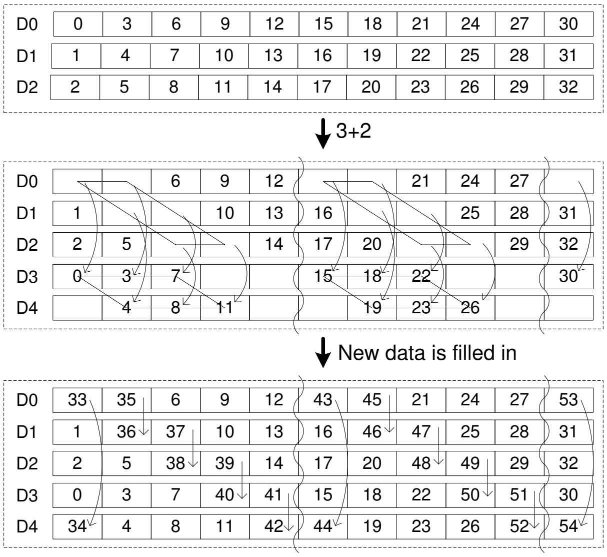

D0 0 3 6 9 12 15 18 21 24 27 30
D1 1 4 7 10 13 16 19 22 25 28 31
D2 2 5 8 11 14 17 20 23 26 29 32
3+2
D0 6 9 12 21 24 27
D1 1 10 13 16 25 28 31
D2 2 5 14 17 20 29 32
D3 0 3 7 15 18 22 30
D4 4 8 11 19 23 26
New data is filled in
D0 33 35 6 9 12 43 45 21 24 27 53
D1 1 36 37 10 13 16 46 47 25 28 31
D2 2 5 38 39 14 17 20 48 49 29 32
D3 0 3 7 40 41 15 18 22 50 51 30
D4 34 4 8 11 42 44 19 23 26 52 54

Figure 2: RAID scaling from 3 disks to 5 using FastScale, where $m \geq n$.

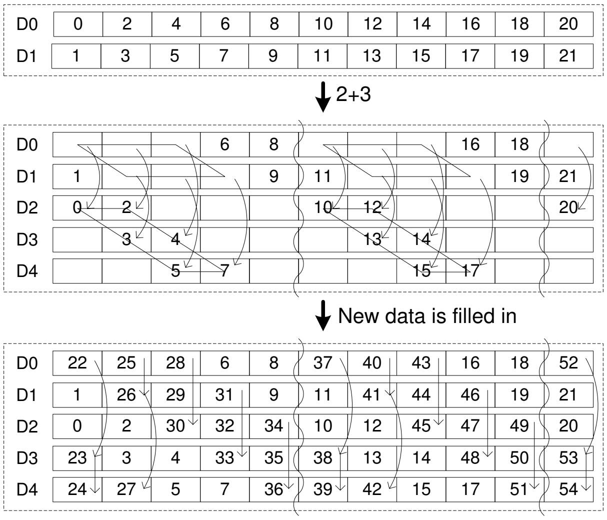

D0 0 2 4 6 8 10 12 14 16 18 20
D1 1 3 5 7 9 11 13 15 17 19 21
2+3
D0 6 8 16 18
D1 1 9 11 19 21
D2 0 2 10 12 20
D3 3 4 13 14
D4 5 7 15 17
New data is filled in
D0 22 25 28 6 8 37 40 43 16 18 52
D1 1 26 29 31 9 11 41 44 46 19 21
D2 0 2 30 32 34 10 12 45 47 49 20
D3 23 3 4 33 35 38 13 14 48 50 53
D4 24 27 5 7 36 39 42 15 17 51 54

Figure 3: RAID scaling from 2 disks to 5 using FastScale, where $m < n$.

into one *region*. In Figure 2, different regions are separated with a wavy line. For different regions, the ways to data migration and data filling are exactly identical.

In a region, all of the data blocks within a parallelogram will be moved. The base of the parallelogram is 2, and the height is 3. In other words, 2 data blocks are selected from each old disk and migrated to new disks. The 2 blocks are sequential, and the start address is *disk_no*. Figure 2 depicts the moving trace of each migrating block. For one moving data block, only its physical disk number is changed while its physical block number is unchanged. As a result, the five columns of two new disks will contain 1, 2, 2, 1, and 0 migrated data blocks, respectively. Here, the data block in the first column will be placed upon disk 3, while the data block in the fourth column will be placed upon disk 4. The first blocks in columns 2 and 3 are placed on disk 3, and the second blocks in columns 2 and 3 are placed on disk 4. Thus, each new disk has 3 data blocks.

After data migration, each disk, either old or new, has 3 data blocks. That is to say, FastScale regains a uniform data distribution. The total number of data blocks to be moved is $2 \times 3 = 6$. This reaches the minimal number of moved blocks, $(5 \times 3) \times (2/(3+2)) = 6$. We can claim that the RAID scaling using FastScale can satisfy Requirement 1 and Requirement 2.

Let us examine whether FastScale can satisfy Requirement 3, i.e., fast data addressing. To consider how one logical data block is addressed, we divide all the data space in the RAID into three categories: original and unmoved data, original and migrated data, and new data. A conclusion can be drawn from the following description that the calculation overhead for the data addressing is very low.

- The original and unmoved data can be addressed with the original addressing method. In this example, the ordinal number of the disk holds one block x can be calculated: $d = x \mod 3$. Its physical block number can be calculated: $b = x/3$.

- The addressing method for original and migrated data can be obtained easily from the above description about the trace of the data migration. $b = x/3$. For those blocks in the first triangle, i.e., blocks 0, 3, and 4, we have $d = d_0 + 3$. For those blocks in the last triangle, i.e., blocks 7, 8, and 11, we have $d = d_0 + 2$. Here, d_0 is their original disk.

- Each region can hold $5 \times 2 = 10$ new blocks. In one region, how those new data blocks are placed is shown in Figure 2. If block x is a new block, it is the y^{th} new block, where $y = x - 3 \times 11$. Each stripe holds 2 new blocks. So, we have $b = y/2$. The first two new blocks in each region are placed on Blocks 0 of Disk 0 and 4. For the other blocks, $d = (y \bmod 2) + (b \bmod 5) - 1$.

Example 2: In the above example, the number of the old disks m and the number of the new disks n satisfy the condition: $m \geq n$. In the following, we inspect the case when $m < n$. Take RAID scaling from 2 disks to 5 as an example. Here, $m = 2$ and $n = 3$.

Likewise, in a region, all of the data blocks within a parallelogram will be moved. The base of the parallelogram is 3, and the height is 2. 3 consecutive data blocks are selected from each old disk and migrated to

```
Algorithm: Addressing(t, H, s, x, d, b)
Input:
    t: scaling times
    H: scaling history, H[0],..., H[t]
    s: total number of data blocks in one disk
    x: logical block number
Output:
    d: the disk holding Block x
    b: physical block number
1: if t = 0 then
2:       $m \leftarrow H[0]$,    $d \leftarrow x \bmod m$,    $b \leftarrow x / m$
3:       exit
4: $m \leftarrow H[t-1]$,    $n \leftarrow H[t] - m$,    $\delta \leftarrow m - H[0]$
5: if $x \in [0, m \times s - 1]$    // an original data block
6:       Addressing(t-1, H, s, x, $d_0$, $b_0$)
7:       $b_1 \leftarrow (b_0 - \delta) \bmod (m+n)$
8:       if $b_1 \in [d_0, d_0 + n - 1]$        // to be moved
9:             $d \leftarrow$ Moving($d_0$, $b_1$, m, n),    $b \leftarrow b_0$
10:       else        // not moved
11:             $d \leftarrow d_0$,    $b \leftarrow b_0$
12:else     // a new data block
13:       Placing(x, m, n, s, $\delta$, d, b)
```

Table 1: The addressing algorithm using in FastScale.

new disks. Figure 3 depicts the trace of each migrating block. Similarly, for one moving data block, only its physical disk number is changed while its physical block number is unchanged. As a result, five columns of three new disks will have a different number of existing data blocks: 1, 2, 2, 1, 0. Here, the data block in the first column will be placed upon disk 3, while the data block in the fourth column will be placed upon disk 4. Unlike the first example, the first block in columns 2 and 3 are placed on disks 2 and 3, respectively. Thus, each new disk has 2 data blocks.

Similar to the first example, we can demonstrate that the RAID scaling using FastScale can satisfy the three requirements.

2.3 The Addressing Algorithm

Table 1 shows the algorithm to minimize data migration required by RAID scaling. The array *H* records the history of RAID scaling. *H[0]* is the initial number of disks in the RAID. After the i^{th} scaling operations, the RAID consists of *H[i]* disks.

When a RAID is constructed from scratch (i.e., $t = 0$), it is a round-robin RAID actually. The address of block *x* can be calculated via one division and one modular (line 2).

Let us inspect the t^{th} scaling, where *n* disks are added into a RAID made up of *m* disks (line 4).

(1) If block *x* is an original block (line 5), FastScale calculates its old address (d_0, b_0) before the t^{th} scaling (line 6).

```
Function: Moving($d_0$, $b_1$, m, n)
Input:
    $d_0$: the disk of the original location
    $b_1$: the original location in a region
    m: the number of old disks
    n: the number of new disks
Output:
    return value: new disk holding the block
1:   if $m \geq n$
2:         if $b_1 \leq n-1$
3:               return $d_0$+m
4:         if $b_1 \geq m-1$
5:               return $d_0$+n
6:         return m+n-1- ($b_1$-$d_0$)
7:   if $m < n$
8:         if $b_1 \leq m-1$
9:               return $d_0$+m
10:        if $b_1 \geq n-1$
11:              return $d_0$+n
12:        return $d_0$+ $b_1$+1
```

Table 2: The Moving function.

- If (d_0, b_0) needs to be moved, FastScale changes the disk ordinal number while keeping the block ordinal number unchanged (line 9).
- If (d_0, b_0) does not need to be moved, FastScale keeps the disk ordinal number and the block ordinal number unchanged (line 11).

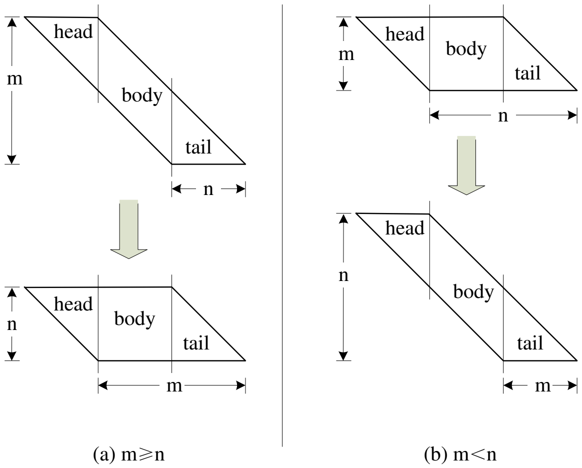
(2) If block *x* is a new block, FastScale places it via the Placing() procedure (line 13).

The code of line 8 is used to decide whether a data block (d_0, b_0) will be moved during RAID scaling. As shown in Figures 2 and 3, there is a parallelogram in each region. The base of the parallelogram is *n*, and the height is *m*. If and only if the data block is within a parallelogram, it will be moved. One parallelogram mapped to disk d_0 is a line segment. Its beginning and end are d_0 and $d_0 + n - 1$, respectively. If b_1 is within the line segment, block *x* is within the parallelogram, and therefore it will be moved. After a RAID scaling by adding *n* disks, the left-above vertex of the parallelogram proceeds by *n* blocks (line 7).

Once a data block is determined to be moved, FastScale changes its disk ordinal number with the Moving() function. As shown in Figure 4, a migrating parallelogram is divided into three parts: a head triangle, a body parallelogram, and a tail triangle. How a block moves depends on which part it lies in. No matter which is bigger between *m* and *n*, the head triangle and the tail

```
Procedure: Placing(x, m, n, s, δ, d, b)
Input:
    x: logical block number
    m: the number of old disks
    n: the number of new disks
    s: total number of data blocks in one disk
    δ: offset of the first region
Output:
    d: new disk holding the block
    b: physical block of new location
1: $y \leftarrow x - m \times s$
2: $b \leftarrow y / n$        $row \leftarrow y \bmod n$
3: $e \leftarrow (b - \delta) \bmod (m+n)$
4: if    $e < n$
5:       if $row < e+1$
6:            $d \leftarrow row$
7:       else
8:            $d \leftarrow row+m$
9: else
10:      $d \leftarrow row+e-n+1$
```

Table 3: The procedure to place new data.



(a) m⩾n

(b) m<n

Figure 4: The variation of data layout involved in migration.

triangle keep their shapes unchanged. The head triangle will be moved by *m* disks (line 3, 9), while the tail triangle will be moved by *n* disks (line 5, 11). However, the body is sensitive to the relationship between *m* and *n*. The body is twisted from a parallelogram to a rectangle when $m \geq n$ (line 6), while from a rectangle to a parallelogram when $m < n$ (line 12). FastScale keeps the relative locations of all blocks in the same column.

When block *x* is in the location newly added after the last scaling, it is addressed via the Placing() procedure. If block *x* is a new block, it is the y^{th} new block (line 1). Each stripe holds *n* new blocks. So, we have $b = y/n$ (line 2). The order of placing new blocks is shown in Figures 2 and 3 (line 4-10).

This algorithm is very simple. It requires fewer than 50 lines of C code, reducing the likelihood that a bug will cause a data block to be mapped to the wrong location.

2.4 Property Examination

The purpose of this experiment is to quantitatively characterize whether the FastScale algorithm satisfies the three requirements, described in Subsection 2.1. For this purpose, we compare FastScale with the round-robin algorithm and the semi-RR algorithm. From a 4-disk array, we add one disk repeatedly for 10 times using the three algorithms respectively. Each disk has a capacity of 128 GB, and the size of a data block is 64 KB. In other words, each disk holds 2×1024^2 blocks.

Uniform data distribution. We use the coefficient of variation as a metric to evaluate the uniformity of data distribution across all the disks. The coefficient of variation expresses the standard deviation as a percentage of the average. The smaller the coefficient of variation is, the more uniform the data distribution is. Figure 5 plots the coefficient of variation versus the number of scaling operations. For the round-robin and FastScale algorithms, both the coefficients of variation remain 0 percent as the times of disk additions increases.

Conversely, the semi-RR algorithm causes excessive oscillation in the coefficient of variation. The maximum is even 13.06 percent. The reason for this non-uniformity is given as follows. An initial group of 4 disks makes the blocks be placed in a round-robin fashion. When the first scaling operation adds one disk, then 1/5 of all blocks, where $(x \bmod 5) \geq 4$, are moved onto the new disk, Disk 4. However, with another operation of adding one more disk using the same approach, 1/6 of all the blocks are not evenly picked from the 5 old disks and moved onto the new disk, Disk 5. Only certain blocks from disks 1, 3 and 4 are moved onto disk 5 while disk 0 and disk 2 are ignored. This is because disk 5 will contain blocks with logical numbers that satisfy $(x \bmod 6) = 5$, which are all odd numbers. The logical numbers of those blocks on Disks 0 and 2, resulting from $(x \bmod 4) = 0$ and $(x \bmod 4) = 2$ respectively, are all even numbers. Therefore, blocks from disks 0 and 2 do not qualify and are not moved.

Minimal data migration. Figure 6 plots the migration fraction (i.e., the fraction of data blocks to be migrated) versus the number of scaling operations. Using the round-robin algorithm, the migration fraction is constantly 100%. This will bring a very large migration cost.

The migration fractions using the semi-RR algorithm and using FastScale are identical. They are significantly smaller than the migration fraction of using the round-

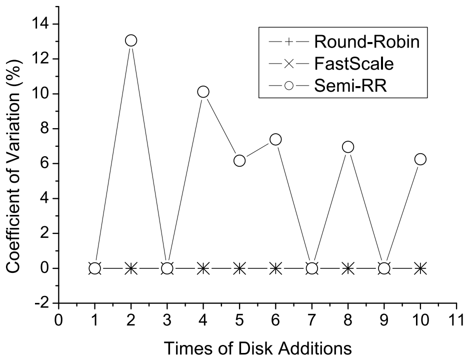


Figure 5: Comparison in uniform data distribution

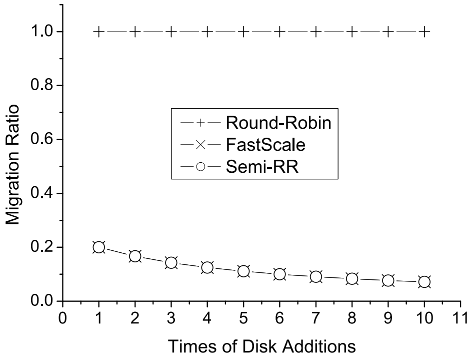


Figure 6: Comparison in data migration ratio

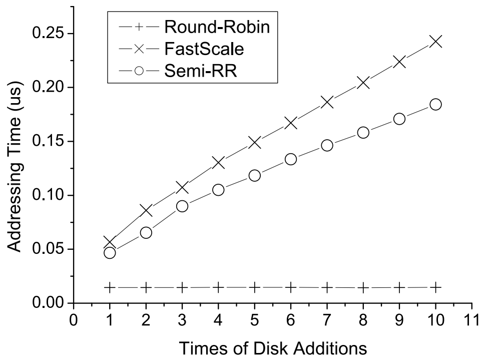


Figure 7: Comparison in addressing time

Algorithm	Storage Overhead
round-robin	1
semi-RR	*t*
FastScale	*t*

Table 4: The storage overheads of different algorithms.

robin algorithm. Another obvious phenomenon is that they decrease with the increase of the number of scaling operations. The reason behind this phenomenon is described as follows. To make each new disk hold $1/(m+n)$ of total data, the semi-RR algorithm and FastScale moves $n/(m+n)$ of total data. m increases with the number of scaling operations. As a result, the percentage of new disks (i.e., $n/(m+n)$) decreases. Therefore, the migration fractions using the semi-RR algorithm and FastScale decrease.

Storage and calculation overheads. When a disk array boots, it needs to obtain the RAID topology from disks. Table 4 shows the storage overheads of the three algorithms. The round-robin algorithm depends only on the total number of member disks. So its storage overhead is one integer. The semi-RR and FastScale algorithms depend on how many disks are added during each scaling operation. If we scale RAID t times, their storage overheads are t integers. Actually, the RAID scaling operation is not too frequent. It may be performed every half year, or even longer. Consequently, the storage overheads are very small.

To quantitatively characterize the calculation overheads, we run different algorithms to calculate the physical addresses for all data blocks on a scaled RAID. The whole addressing process is timed and then the average addressing time for each block is calculated. The testbed used in the experiment is an Intel Dual Core T9400 2.53 GHz machine with 4 GB of memory. A Windows 7 Enterprise Edition is installed. Figure 7 plots the addressing time versus the number of scaling operations.

The round-robin algorithm has a low calculation overhead of 0.014 μs or so. The calculation overheads using the semi-RR and FastScale algorithms are close, and both take on an upward trend. Among the three algorithms, FastScale has the largest overhead. Fortunately, the largest addressing time using FastScale is 0.24 μs which is negligible compared to milliseconds of disk I/O time.

3 Optimizing Data Migration

The FastScale algorithm succeeds in minimizing data migration for RAID scaling. In this section, we describe FastScale's optimizations to the process of data migration.

3.1 Access Aggregation

FastScale moves only data blocks from old disks to new disks, while not migrating data among old disks. The data migration will not overwrite any valid data. As a result, data blocks may be moved in an arbitrary order. Since disk I/O performs much better with large sequential access, FastScale accesses multiple successive blocks via a single I/O.

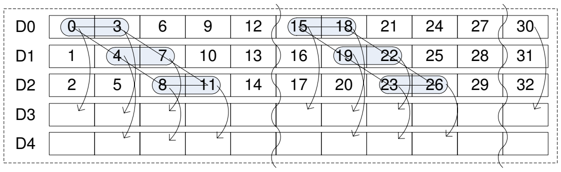

D0 0 3 6 9 12 15 18 21 24 27 30
D1 1 4 7 10 13 16 19 22 25 28 31
D2 2 5 8 11 14 17 20 23 26 29 32
D3
D4

Figure 8: Aggregate reads for RAID scaling from 3 disks to 5. Multiple successive blocks are read via a single I/O.

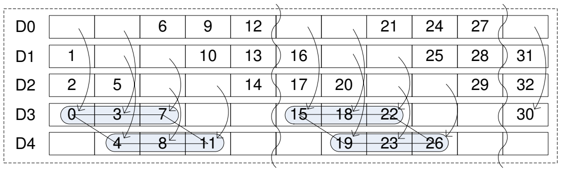


Figure 9: Aggregate writes for RAID scaling from 3 disks to 5. Multiple successive blocks are written via a single I/O.

Take a RAID scaling from 3 disks to 5 as an example, shown in Figure 8. Let us focus on the first region. FastScale issues the first I/O request to read Blocks 0 and 3, the second request to read Blocks 4 and 7, and the third request for Blocks 8 and 11, simultaneously. By this means, to read all of these blocks, FastScale requires only three I/Os, instead of six. Furthermore, all these 3 large-size data reads are on three disks. They can be done in parallel, further increasing I/O rate.

When all the six blocks have been read into a memory buffer, FastScale issues the first I/O request to write Blocks 0, 3, and 7, the second I/O to write Blocks 4, 8 and 11, simultaneously (see Figure 9). In this way, only two large sequential write requests are issued as opposed to six small writes.

For RAID scaling from *m* disks to *m*+*n*, *m* reads and *n* writes are required to migrate all the data in a region, i.e., $m \times n$ data blocks.

Access aggregation converts sequences of small requests into fewer, larger requests. As a result, seek cost is mitigated over multiple blocks. Moreover, a typical choice of the optimal block size for RAID is 32KB or 64KB [4, 7, 8, 9]. Thus, accessing multiple successive blocks via a single I/O enables FastScale to have a larger throughput. Since data densities in disks increase at a much faster rate than improvements in seek times and rotational speeds, access aggregation benefits more as technology advances.

3.2 Lazy Checkpoint

While data migration is in progress, the RAID storage serves user requests. Furthermore, the coming user I/Os may be write requests to migrated data. As a result, if mapping metadata does not get updated until all of the blocks have been moved, data consistency may be destroyed. Ordered operations [9] of copying a data block and updating the mapping metadata (a.k.a., *checkpoint*) can ensure data consistency. But ordered operations cause each block movement to require one metadata write, which results in a large cost of data migration. Because metadata is usually stored at the beginning of all member disks, each metadata update causes one long seek per disk. FastScale uses lazy checkpoint to minimize the number of metadata writes without compromising data consistency.

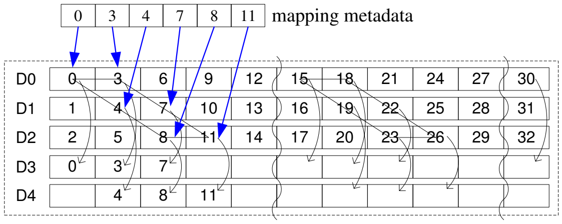

0 3 4 7 8 11 mapping metadata
D0 0 3 6 9 12 15 18 21 24 27 30
D1 1 4 7 10 13 16 19 22 25 28 31
D2 2 5 8 11 14 17 20 23 26 29 32
D3 0 3 7
D4 4 8 11

Figure 10: If data blocks are copied to their new locations and metadata is not yet updated when the system fails, data consistency is still maintained because the data in their original locations are valid and available.

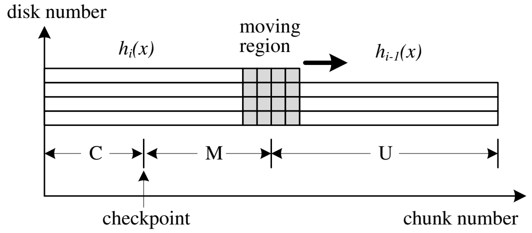


Figure 11: Lazy updates of mapping metadata. “C”: migrated and checkpointed; “M”: migrated but not checkpointed; “U”:not migrated. Data redistribution is checkpointed only when a user write request arrives in the area “M”.

The foundation of lazy checkpoint is described as follows. Since block copying does not overwrite any valid data, both its new replica and original are valid after a data block is copied. In the above example, we suppose that Blocks 0, 3, 4, 7, 8, and 11 have been copied to their new locations and the mapping metadata has not been updated (see Figure 10), when the system fails. The original replicas of the six blocks will be used after the system reboots. As long as Blocks 0, 3, 4, 7, 8, and 11 have not been written since being copied, the data remain consistent. Generally speaking, when the mapping information is not updated immediately after a data block is copied, an unexpected system failure only wastes some data accesses, but does not sacrifice data reliability. The only threat is the incoming of write operations to migrated data.

The key idea behind lazy checkpoint is that data blocks are copied to new locations continuously, while the mapping metadata is not updated onto the disks (a.k.a., *checkpoint*) until a threat to data consistency appears. We use $h_i(x)$ to describe the geometry after the i^{th} scaling opera-

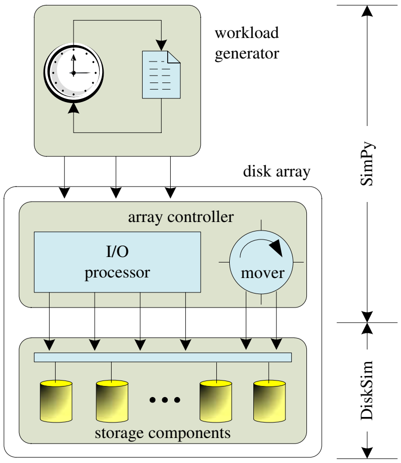


Figure 12: Simulation system block diagram: The workload generator and the array controller were implemented in SimPy. DiskSim was used as a worker module to simulate disk accesses.

tion, where N_i disks serve user requests. Figure 11 illustrates an overview of the migration process. Data in the moving region is copied to new locations. When a user request arrives, if its physical block address is above the moving region, it is mapped with $h_{i-1}(x)$; If its physical block address is below the moving region, it is mapped with $h_i(x)$. When all of the data in the current moving region are moved, the next region becomes the moving region. In this way, the newly added disks are gradually available to serve user requests. Only when a user write request arrives in the area where data have been moved and the movement has not been checkpointed, are mapping metadata updated.

Since one write of metadata can store multiple map changes of data blocks, lazy updates can significantly decrease the number of metadata updates, reducing the cost of data migration. Furthermore, lazy checkpoint can guarantee data consistency. Even if the system fails unexpectedly, only some data accesses are wasted. It should also be noted that the probability of a system failure is very low.

4 Experimental Evaluation

The experimental results in Section 2.4 show that the semi-RR algorithm causes extremely non-uniform data distribution. This will bring into low I/O performance. In this section, we compare FastScale with the SLAS approach [5] through detailed experiments. SLAS, proposed in 2007, preserves the round-robin order after adding disks.

4.1 Simulation System

We use detailed simulations with several disk traces collected in real systems. The simulator is made up of a workload generator and a disk array (Figure 12). According to trace files, the workload generator initiates an I/O request at the appropriate time so that a particular workload is induced on the disk array.

The disk array consists of an array controller and storage components. The array controller is logically divided into two parts: an I/O processor and a data mover. The I/O processor, according to the address mapping, forwards incoming I/O requests to the corresponding disks. The data mover reorganizes the data on the array. The mover uses an on/off logic to adjust the redistribution rate. Data redistribution is throttled on detection of high application workload. Otherwise, it performs continuously.

The simulator is implemented in SimPy [10] and DiskSim [11]. SimPy is an object-oriented, process-based discrete-event simulation language based on standard Python. DiskSim is an efficient, accurate disk system simulator from Carnegie Mellon University and has been extensively used in various research projects studying storage subsystem architectures. The workload generator and the array controller are implemented in SimPy. Storage components are implemented in DiskSim. In other words, DiskSim is used as a worker module to simulate disk accesses. The simulated disk specification is that of the 15,000-RPM IBM Ultrastar 36Z15 [12].

4.2 Workloads

Our experiments use the following three real-system disk I/O traces with different characteristics.

- *TPC-C* traced disk accesses of the TPC-C database benchmark with 20 warehouses [13]. It was collected with one client running 20 iterations.
- *Fin* is obtained from the Storage Performance Council (SPC) [14, 15], a vendor-neutral standards body. The Fin trace was collected from OLTP applications running at a large financial institution. The write ratio is high.
- *Web* is also from SPC. It was collected from a system running a web search engine. The read-dominated Web trace exhibits the strong locality in its access pattern.

4.3 Experiment Results

4.3.1 The Scaling Efficiency

Each experiment lasts from the beginning to the end of data redistribution for RAID scaling. We focus on com-

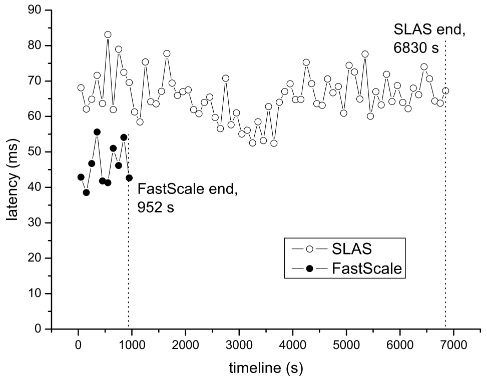


Figure 13: Performance comparison between FastScale and SLAS under the Fin workload.

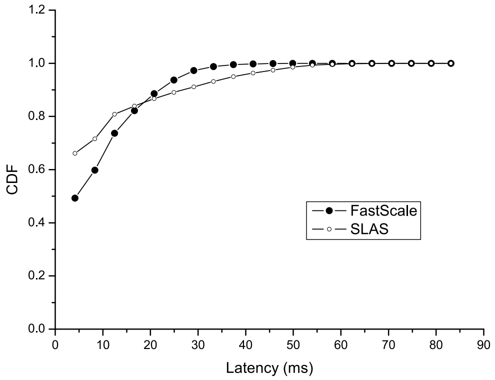


Figure 14: Cumulative distribution of I/O latencies during the data redistributions by the two approaches under the Fin workload.

paring redistribution times and user I/O latencies when different scaling programs are running in background.

In all experiments, the sliding window size for SLAS is set to 1024. Access aggregation in SLAS can improve the redistribution efficiency. However, a too large size of redistribution I/Os will compromise the I/O performance of applications. In our experiments, SLAS reads 8 data blocks via an I/O request.

The purpose of our first experiment is to quantitatively characterize the advantages of FastScale through a comparison with SLAS. We conduct a scaling operation of adding 2 disks to a 4-disk RAID, where each disk has a capacity of 4 GB. Each approach performs with the 32KB stripe unit size under a Fin workload. The threshold of rate control is set 100 IOPS. This parameter setup acts as the baseline for the latter experiments, from which any change will be stated explicitly.

We collect the latencies of all user I/Os. We divide the I/O latency sequence into multiple sections according to I/O issuing time. The time period of each section is 100 seconds. Furthermore, we get a local maximum latency from each section. A local maximum latency is the maximum of I/O latencies in a section. Figure 13 plots local maximum latencies using the two approaches as the time increases along the x-axis. It illustrates that FastScale demonstrates a noticeable improvement over SLAS in two metrics. First, the redistribution time using FastScale is significantly shorter than that using SLAS. They are 952 seconds and 6,830 seconds, respectively. In other words, FastScale has a 86.06% shorter redistribution time than SLAS.

The main factor in FastScale's reducing the redistribution time is the significant decline of the amount of the data to be moved. When SLAS is used, almost 100% of data blocks have to be migrated. However, when FastScale is used, only 33.3% of data blocks require to be migrated. Another factor is the effective exploitation of two optimization technologies: access aggregation reduces the number of redistribution I/Os; lazy checkpoint minimizes metadata writes.

Second, local maximum latencies of SLAS are obviously longer than those of FastScale. The global maximum latency using SLAS reaches 83.12 ms while that using FastScale is 55.60 ms. This is because the redistribution I/O size using SLAS is larger than that using FastScale. For SLAS, the read size is 256 KB (8 blocks), and the write size is 192 KB (6 blocks). For FastScale, the read size is 64 KB (2 blocks), and the write size is 128 KB (4 blocks). Of course, local maximum latencies of SLAS will be lower with a decrease in the redistribution I/O size. But the decrease in the I/O size will necessarily enlarge the redistribution time.

Figure 14 shows the cumulative distribution of user response times during data redistribution. To provide a fair comparison, I/Os involved in statistics for SLAS are only those issued before 952 seconds. When I/O latencies are larger than 18.65 ms, the CDF value of FastScale is larger than that of SLAS. This indicates again that FastScale has smaller maximum response time of user I/Os than SLAS. The average latency of FastScale is close to that of SLAS. They are 8.01 ms and 7.53 ms respectively. It is noteworthy that due to significantly shorter data redistribution time, FastScale has a markedly smaller impact on the user I/O latencies than SLAS does.

A factor that might affect the benefits of FastScale is the workload under which data redistribution performs. Under the TPC-C workload, we also measure the performances of FastScale and SLAS to perform the "4+2" scaling operation.

For the TPC-C workload, Figure 15 shows local maximum latencies versus the redistribution times for SLAS and FastScale. It shows once again the efficiency of FastScale in improving the redistribution time. The redistribution times using SLAS and FastScale are 6,820 seconds and 964 seconds, respectively. That is to say, FastScale brings an improvement of 85.87% in the re-

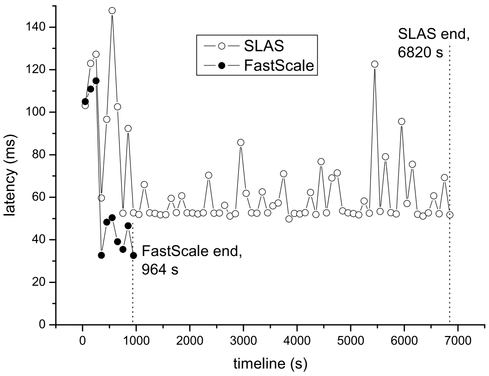


Figure 15: Performance comparison between FastScale and SLAS under the TPC-C workload.

distribution time. Likewise, local maximum latencies of FastScale are also obviously shorter than those of SLAS. The global maximum latency using FastScale is 114.76 ms while that using SLAS reaches 147.82 ms.

To compare the performance of FastScale under different workloads, Figure 16 shows a comparison in the redistribution time between FastScale and SLAS. For completeness, we also conducte a comparison experiment on the redistribution time with no loaded workload. To scale a RAID volume off-line, SLAS uses 6802 seconds whereas FastScale consumes only 901 seconds. FastScale provides an improvement of 86.75% in the redistribution time.

We can draw one conclusion from Figure 16. Under various workloads, FastScale consistently outperformes SLAS by 85.87-86.75% in the redistribution time, with smaller maximum response time of user I/Os.

4.3.2 The Performance after Scaling

The above experiments show that FastScale improves the scaling efficiency of RAID significantly. One of our concerns is whether there is a penalty in the performance of the data layout after scaling using FastScale, compared with the round-robin layout preserved by SLAS.

We use the Web workload to measure the performances of the two RAIDs, scaled from the same RAID using SLAS and FastScale. Each experiment lasts 500 seconds, and records the latency of each I/O. Based on the issue time, the I/O latency sequence is divided into 20 sections evenly. Furthermore, we get a local average latency from each section.

First, we compare the performances of two RAIDs, after one scaling operation "4+1" using the two scaling approaches. Figure 17 plots local average latencies for the two RAIDs as the time increases along the x-axis. We can find that the performances of the two RAIDs are very close. With regards to the round-robin RAID, the average latency is 11.36 ms. For the FastScale RAID, the average latency is 11.37 ms.

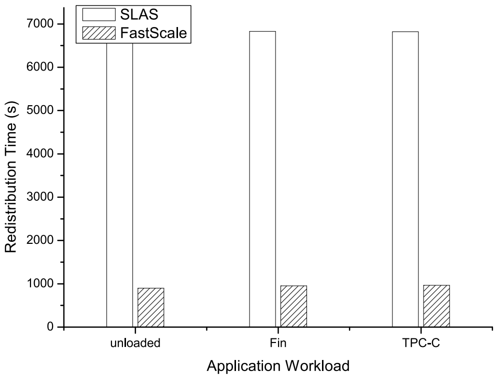


Figure 16: Comparison of redistribution times of FastScale and SLAS under different workloads. The label "unloaded" means scaling a RAID volume offline.

Second, we compare the performances of two RAIDs, after two scaling operations "4+1+1" using the two approaches. Figure 18 plots local average latencies of the two RAIDs as the time increases along the x-axis. It again revealed the approximate equality in the performances of the two RAIDs. With regards to the round-robin RAID, the average latency is 11.21 ms. For the FastScale RAID, the average latency is 11.03 ms.

One conclusion can be reached that the performance of the RAID scaled using FastScale is almost identical with that of the round-robin RAID.

5 Related Work

5.1 Scaling Deterministic RAID

The HP AutoRAID [8] allows an online capacity expansion. Newly created RAID-5 volumes use all of the disks in the system, but previously created RAID-5 volumes continue to use only the original disks. This expansion does not require data migration. But the system cannot add new disks into an existing RAID-5 volume. The conventional approaches to RAID scaling redistributes data and preserves the round-robin order after adding disks.

Gonzalez and Cortes [3] proposed a gradual assimilation algorithm (GA) to control the overhead of scaling a RAID-5 volume. However, GA accesses only one block via an I/O. Moreover, it writes mapping metadata onto disks immediately after redistributing each stripe. As a result, GA has a large redistribution cost.

The reshape toolkit in the Linux MD driver (MD-Reshape) [4] writes mapping metadata for each fixed-sized data window. However, user requests to the data window have to queue up until all data blocks within the window are moved. On the other hand, MD-Reshape issues very small (4KB) I/O operations for data redistribution. This limits the redistribution performance due to

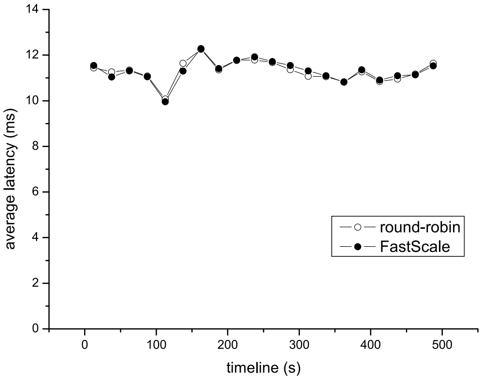


Figure 17: Performance comparison between FastScale's layout and round-robin layout under the Web workload after one scaling operation "4+1".

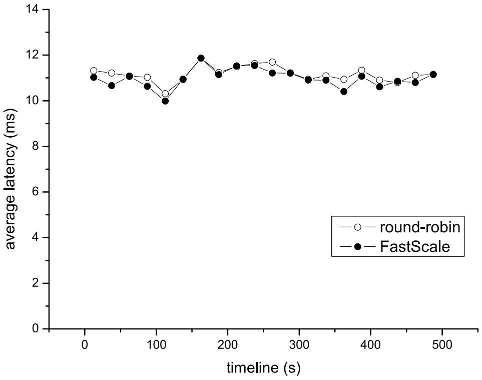


Figure 18: Performance comparison between FastScale's layout and round-robin layout under the Web workload after two scaling operations "4+1+1".

more disk seeks.

Zhang et al. [5] discovered that there is always a reordering window during data redistribution for round-robin RAID scaling. The data inside the reordering window can migrate in any order without overwriting any valid data. By leveraging this insight, they proposed the SLAS approach, improving the efficiency of data redistribution. However, SLAS still requires migrating all data. Therefore, RAID scaling remains costly.

D-GRAID [16] restores only live file system data to a hot spare so as to recover from failures quickly. Likewise, it can accelerate the redistribution process if only the live data blocks from the perspective of file systems are redistributed. However, this needs for semantically-smart storage systems. Differently, FastScale is independent on file systems, and it can work with any ordinary disk storage.

A patent [17] presents a method to eliminate the need to rewrite the original data blocks and parity blocks on original disks. However, the method makes all the parity blocks be either only on original disks or only on new disks. The obvious distribution non-uniformity of parity blocks will bring a penalty to write performance.

Franklin et al. [18] presented an RAID scaling method using spare space with immediate access to new space. First, old data are distributed among the set of data disk drives and at least one new disk drive while, at the same time, new data are mapped to the spare space. Upon completion of the distribution, new data are copied from the spare space to the set of data disk drives. This is similar to the key idea of WorkOut [19]. This kind of method requires spare disks available in the RAID.

In another patent, Hetzler [20] presented a method to RAID-5 scaling, noted MDM. MDM exchanges some data blocks between original disks and new disks. MDM can perform RAID scaling with reduced data movement. However, it does not increase (just maintains) the data storage efficiency after scaling. The RAID scaling process exploited by FastScale is favored in the art because the data storage efficiency is maximized, which many practitioners consider desirable.

5.2 Scaling Randomized RAID

Randomized RAID [6, 21, 22, 23] appears to have better scalability. It is now gaining the spotlight in the data placement area. Brinkmann et al. [23] proposed the cut-and-paste placement strategy that uses randomized allocation strategy to place data across disks. For a disk addition, it cuts off the range $[1/(n+1), 1/n]$ from given n disks, and pastes them to the newly added $(n+1)^{th}$ disk. For a disk removal, it uses reversing operation to move all the blocks in disks that will be removed to the other disks. Also based on random data placement, Seo and Zimmermann [24] proposed an approach to finding a sequence of disk additions and removals for the disk replacement problem. The goal is to minimize the data migration cost. Both these two approaches assume the existence of a high-quality hash function that assigns all the data blocks in the system into the uniformly distributed real numbers with high probability. However, they did not present such a hash function.

The SCADDAR algorithm [6] uses a pseudo-random function to distribute data blocks randomly across all disks. It keeps track of the locations of data blocks after multiple disk reorganizations and minimizes the amount of data to be moved. Unfortunately, the pseudo-hash function does not preserve the randomness of the data layout after several disk additions or deletions [24]. So far, true randomized hash function which preserves its randomness after several disk additions or deletions has not been found.

The simulation report in [21] shows that a single copy of data in random striping may result in some hiccups of the continuous display. To address this issue, one can use data replication [22], where a fraction of the data blocks randomly selected are replicated on randomly selected disks. However, this will bring into a large capacity overhead.

RUSH [25, 26] and CRUSH [27] are two algorithms for online placement and reorganization of replicated data. They are probabilistically optimal in distributing data evenly and minimizing data movement when new storage is added to the system. There are three differences between them and FastScale. First, they depend on the existence of a high-quality random function, which is difficult to generate. Second, they are designed for object-based storage systems. They focus on how a data object is mapped to a disk, without considering the data layout of each individual disk. Third, our mapping function needs to be 1-1 and onto, but hash functions have collisions and count on some amount of sparseness.

6 Conclusion and Future Work

This paper presents FastScale, a new approach that accelerates RAID-0 scaling by minimizing data migration. First, with a new and elastic addressing function, FastScale minimizes the number of data blocks to be migrated without compromising the uniformity of data distribution. Second, FastScale uses access aggregation and lazy checkpoint to optimize data migration.

Our results from detailed experiments using real-system workloads show that, compared with SLAS, a scaling approach proposed in 2007, FastScale can reduce redistribution time by up to 86.06% with smaller maximum response time of user I/Os. The experiments also illustrate that the performance of the RAID scaled using FastScale is almost identical with that of the round-robin RAID.

In this paper, the factor of data parity is not taken into account. we believe that FastScale provides a good starting point for efficient scaling of RAID-4 and RAID-5 arrays. In the future, we will focus on extending FastScale to RAID-4 and RAID-5.

7 Acknowledgements

We are indebted to the anonymous reviewers of the paper for their insightful comments. We are also grateful to Dr. Benjamin Reed, our shepherd, for detailed comments and suggestions that greatly improved the readability of the paper. This work was supported by the National Natural Science Foundation of China under Grant 60903183, the National High Technology Research and Development Program of China under Grant No. 2009AA01A403, and the National Grand Fundamental Research 973 Program of China under Grant No. 2007CB311100.

References

[1] D. A. Patterson, G. A. Gibson, R. H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID), in Proceedings of the International Conference on Management of Data (SIGMOD'88), June 1988. pp. 109-116.

[2] D. A. Patterson. A simple way to estimate the cost of down-time. In Proceedings of the 16th Large Installation Systems Administration Conference (LISA'02), October 2002. pp. 185-188.

[3] J. Gonzalez and T. Cortes. Increasing the capacity of RAID5 by online gradual assimilation. In Proceedings of the International Workshop on Storage Network Architecture and Parallel I/Os. Antibes Juan-les-pins, France, Sept. 2004

[4] N. Brown. Online RAID-5 resizing. drivers/md/ raid5.c in the source code of Linux Kernel 2.6.18. http://www.kernel.org/. September 2006.

[5] G. Zhang, J. Shu, W. Xue, and W. Zheng. SLAS: An efficient approach to scaling round-robin striped volumes. ACM Trans. Storage, volume 3, issue 1, Article 3, 1-39 pages. March 2007.

[6] A. Goel, C. Shahabi, S-YD Yao, R. Zimmermann. SCADDAR: An efficient randomized technique to reorganize continuous media blocks. In Proceedings of the 18th International Conference on Data Engineering (ICDE'02). San Jose, 2002. pp. 473-482.

[7] J. Hennessy and D. Patterson. Computer Architecture: A Quantitative Approach, 3rd ed. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 2003.

[8] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. ACM Transactions on Computer Systems, volume 14, issue 1, pp. 108-136, February 1996.

[9] C. Kim, G. Kim, and B. Shin. Volume management in SAN environment. In Proceedings of the 8th International Conference on Parallel and Distributed Systems, ICPADS'01. 2001. pp. 500-505.

[10] Klaus Muller, Tony Vignaux. SimPy 2.0.1's documentation. http://simpy.sourceforge.net/SimPyDocs/index.html. last accessed on April, 2009.

[11] J. Bucy, J. Schindler, S. Schlosser, G. Ganger. The DiskSim Simulation Environment Version 4.0 Reference Manual. Tech. report CMU-PDL-08-101, Carnegie Mellon University. 2008.

[12] Hard disk drive specifications Ultrastar 36Z15. http://www.hitachigst.com/tech/techlib.nsf/techdocs/ 85256AB8006A31E587256A7800739FEB/$file/U36Z15_sp10.PDF. Revision 1.0, April, 2001.

[13] TPC-C. Postgres. 20 iterations. DTB v1.1. Performance Evaluation Laboratory, Brigham Young University. Trace distribution center. http://tds.cs.byu.edu/tds/, last accessed on December, 2010.

[14] OLTP Application I/O and Search Engine I/O. UMass Trace Repository. http://traces.cs.umass.edu/index.php/Storage/Storage. June, 2007.

[15] Storage Performance Council. http://www.storageperformance.org/home. last accessed on December, 2010.

[16] Muthian Sivathanu , Vijayan Prabhakaran , Andrea C. Arpaci-Dusseau , Remzi H. Arpaci-Dusseau. Improving Storage System Availability with D-GRAID, In Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST'04), San Francisco, CA. March 2004.

[17] C.B. Legg, Method of Increasing the Storage Capacity of a Level Five RAID Disk Array by Adding, in a Single Step, a New Parity Block and N-1 New Data Blocks Which Respectively Reside in a new Columns, Where N Is at Least Two, US Patent: 6000010, December 1999.

[18] C.R Franklin and J.T. Wong, Expansion of RAID Subsystems Using Spare Space with Immediate Access to New Space, US Patent 10/033,997, 2006.

[19] Suzhen Wu, Hong Jiang, Dan Feng, Lei Tian, and Bo Mao, WorkOut: I/O Workload Outsourcing for Boosting the RAID Reconstruction Performance, In Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST '09), San Francisco, CA, USA, pp. 239-252. February 2009.

[20] S.R. Hetzler, Data Storage Array Scaling Method and System with Minimal Data Movement, US Patent 20080276057, 2008.

[21] J. Alemany and J. S. Thathachar. Random striping news on demand servers. Tech. Report, TR-97-02-02, University of Washington, 1997.

[22] Jose Renato Santos, Richard R. Muntz, and Berthier A. Ribeiro-Neto. Comparing random data allocation and data striping in multimedia servers. In Measurement and Modeling of Computer Systems, pp. 44-55. 2000.

[23] Andre Brinkmann, Kay Salzwedel, and Christian Scheideler. Efficient, distributed data placement strategies for storage area networks (extended abstract). In ACM Symposium on Parallel Algorithms and Architectures, pp. 119-128. 2000.

[24] Beomjoo Seo and Roger Zimmermann. Efficient disk replacement and data migration algorithms for large disk subsystems. ACM Transactions on Storage (TOS), volume 1, issue 3, pages 316-345, August 2005.

[25] R. J. Honicky and E. L. Miller. A fast algorithm for online placement and reorganization of replicated data. In Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS 2003), Nice, France, April 2003.

[26] R. J. Honicky and E. L. Miller. Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution. In Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04), IEEE. 2004

[27] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data. In Proceedings of the International Conference on Super Computing (SC'06). Tampa Bay, FL. 2006.

The SCADS Director: Scaling a Distributed Storage System Under Stringent Performance Requirements

Beth Trushkowsky, Peter Bodík, Armando Fox,
Michael J. Franklin, Michael I. Jordan, David A. Patterson
{*trush, bodik, fox, franklin, jordan, pattrsn*}@*eecs.berkeley.edu*
University of California, Berkeley

Abstract

Elasticity of cloud computing environments provides an economic incentive for automatic resource allocation of stateful systems running in the cloud. However, these systems have to meet strict performance Service-Level Objectives (SLOs) expressed using upper percentiles of request latency, such as the 99th. Such latency measurements are very noisy, which complicates the design of the dynamic resource allocation. We design and evaluate the SCADS Director, a control framework that reconfigures the storage system on-the-fly in response to workload changes using a performance model of the system. We demonstrate that such a framework can respond to both unexpected data hotspots and diurnal workload patterns without violating strict performance SLOs.

1 Introduction

Cloud computing has emerged as a preferred technology for delivering large-scale internet applications, in part because its elasticity provides the ability to dynamically provision and reclaim resources in response to fluctuations in workload. As cloud environments and their applications expand in scale and complexity, it becomes increasingly important to automate such dynamic resource allocation.

Techniques for automatically scaling stateless systems such as web servers or application servers are fairly well understood. However, many applications that can most benefit from elasticity, such as social networking, e-commerce and auction sites, are both data-intensive and interactive. Such applications present three major challenges for automatic scaling.

First, in most data-intensive services, a request for a specific data item can only be satisfied by a copy of that particular data item, so not every server can handle every request, which complicates load balancing. Second, interactivity means that a successful application must provide highly-responsive, low-latency service to the vast majority of users: a typical Service Level Objective (SLO) might be expressed as "99% of all requests must be answered within 100ms" [20, 17]. Third, the workloads presented by large-scale applications can be highly volatile, with quickly-occurring unexpected spikes (due to flash crowds) and diurnal fluctuations.

This "perfect storm" of statefulness, workload volatility and stringent performance requirements complicates the development of automatic scaling mechanisms. To scale a data-intensive system, data items must be moved (i.e., partitioned or coalesced) or copied (i.e., replicated) among the nodes of the system. Such data movement takes time and can place additional load on an already overloaded system. Provisioning of new nodes incurs significant start-up delay, so decisions must be made early to react effectively to workload changes. But most importantly, the SLOs on upper percentile latency significantly complicate the problem compared to requirements based on average latency, as statistical estimates based on observations in the upper percentiles of the latency distribution have higher variance than estimates obtained from the center of the distribution. This variance is exacerbated by "environmental" application noise uncorrelated to particular queries or data items [19]. The resulting noisy latency signal can cause oscillations in classical closed-loop control [7].

In this paper we describe the design of a control framework for dynamically scaling distributed storage systems that addresses these challenges. Our approach leverages key features of modern distributed storage systems and uses a performance model coupled with workload statistics to predict whether each server is likely to continue to meet its SLO. Based on this model, the framework moves and replicates data as necessary. In particular, we make the following contributions:

- We identify the challenges and opportunities that arise in designing dynamic resource allocation frameworks for stateful systems that maintain performance SLOs on upper quantiles of request latency.

- We describe the design and implementation of a modular control framework based on Model-Predictive Control [30] that addresses these challenges.
- We evaluate the effectiveness of the control framework through experiments using a storage system running on Amazon's Elastic Compute Cloud (EC2), using workloads that exhibit both periodic and erratic fluctuations comparable to those observed in production systems.

The rest of the paper proceeds as follows. Section 2 describes background and challenges, and Section 3 discusses the design considerations that address those challenges. Related work is in Section 4. Section 5 details the implementation of our control framework, and Section 6 demonstrates experimental results of the control framework using Amazon's EC2. Further discussion is in Section 7, and we remark on future work and conclude in Sections 8 and 9.

2 Scaling Challenges

2.1 Background

We address dynamic resource allocation for distributed storage systems for which the performance SLO is specified using an upper percentile of latency. The goal is to design a *control framework* that tries to avoid SLO violations, while keeping the cost of leased resources low.

Our solution is targeted for storage systems designed for horizontal scalability, such as key-value stores, that back interactive web applications. Examples of such systems are PNUTS [17], BigTable [14], Cassandra [3], SCADS [6], and HBase [4]. Requests in these systems have a simple communication pattern; each system at minimum provides `get` and `put` functionality on keys, and each request is single unit of work. We take advantage of this simplicity in our approach.

This simplified model also lends itself to easy partitioning of the key space across multiple servers, typically using a hash or range partitioning scheme. Each server node stores a subset of the data and serves requests for that subset. The control framework has two knobs: it can partition or replicate data to prevent servers from being overloaded when workload increases (e.g. due to diurnal variation or hotspots), or it can coalesce data and remove unnecessary replicas when the workload decreases. To make these configuration changes, the underlying storage system must be easy to reconfigure on-the-fly. Specifically, we require that it allows data to be copied from one server to another or deleted from a server, and that it provides methods like `AddServer` and `RemoveServer` to alter the number of leased servers. We previously designed and built SCADS [6] to both support this functionality and provide the simple communication pattern described above. As we further discuss in Section 7, running our own key-value store in the cloud has advantages over using a cloud-provided data service such as Amazon's S3.

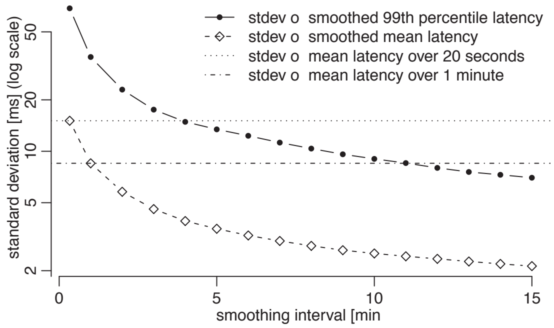


Figure 1: Standard deviation for the mean and 99th percentile of latency for increasing smoothing window sizes. The left-most points represent the raw measurements over 20-second periods. The average of the mean and 99th percentile latencies are 11 ms and 82 ms, respectively.

SCADS was designed to keep data memory-resident so that applications aren't required to use ad-hoc caching techniques to reach performance goals. This design provides similar performance benefits as Memcached; however, SCADS also supports real-time replication and load balancing. An example target application would be the highly interactive social networking site Facebook.com; most of their data remains memory-resident in order to hit performance targets [31].

In this section, we identify two challenges in scaling a storage system while maintaining a high-percentile SLO: noise and data movement. Benchmarks are presented to show the effects of each of these challenges.

2.2 Controlling a Noisy Signal

Figure 1 shows request latencies achieved by several key-value storage servers under a steady workload.[1] As expected, the standard deviation of both the mean latency and 99th percentile latency decreases as we increase the *smoothing window*, or time period over which the measurements are aggregated. However, as can be seen in the figure, the 99th percentile of latency would have be to smoothed over a four-minute window to achieve the same standard deviation as that achieved by the mean smoothed over a 20-second window (an 11x longer smoothing window). Similar effects are illustrated in experiments with Dynamo [20][2].

This observation has serious consequences if we are

[1] The workload consists of `get` and `put` requests against the SCADS [6] storage system, running on ten Amazon Elastic Compute Cloud (EC2) "Small" instances. Details of our experimental setup are in Section 6.1.

[2] See Figure 4 in [20]

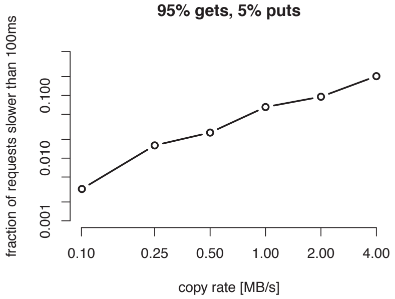


Figure 2: Impact on read performance during data copying on the write target. The x-axis represents the copy rate (in log scale) and the y-axis represents the fraction of requests slower than 100 ms (in log scale).

contemplating using classical closed-loop control. A long smoothing window means a longer delay before the control loop can make its next decision, resulting in more SLO violations. Furthermore, too much smoothing could mask a real spike in workload, and the controller would not respond at all. A short smoothing window mitigates both problems but can lead to oscillatory behavior [7]. Due to the high variance associated with a shorter smoothing window, the controller cannot tell if a server with high latency is actually overloaded or if it is simply exhibiting normally-occurring higher latency. A classical closed-loop controller might add servers in one iteration just to remove them in the next or may move data back and forth unnecessarily in response to such "false alarms." We show in Section 3 that a more effective approach is a *model-based* control in which the controller uses a different input signal than the quantity it is trying to control.

2.3 Data Movement Hurts Performance

Scaling a storage system requires data movement. Because each server is responsible for its own *state*, i.e., the data it stores, it is not generally true that any server can service any request. Simply adding and removing servers is not sufficient to respond to changes in workload, we additionally need to copy and move data between servers. However, data movement impacts performance and this impact is especially noticeable in the tail of the latency distribution. Impacting the tail of the distribution is of particular interest since we target upper percentile SLOs. As demonstrated in Figure 2, copying data increases the fraction of slow requests. In Dynamo [20], the data copy operations are run in low priority mode to minimize their impact on performance of interactive operations. Since one of our operational goals is to respond to spikes while minimizing SLO violations, our approach instead identifies and copies the smallest amount of data needed to relieve SLO pressure.

3 Design Techniques and Approach

Having outlined our goals and identified key challenges in Section 2, we now describe the design techniques in our solution. In particular, we use a model-predictive control, fine-grained workload statistics, and replication for performance predictability.

3.1 Model-Predictive Control

Model-predictive control (MPC) can yield improvements over classical closed-loop control systems in the presence of noisy signals because the controller takes as input a different signal than the one it is trying to control. In MPC, the controller uses a model of the system and its current state to compute the (near) optimal sequence of actions that maintain desired constraints. To simplify the computation of these actions, MPC considers a short receding time horizon. The controller executes only the first action in the sequence and then uses the new current state to compute a new sequence of actions. In each iteration, the controller reevaluates the system state and computes a new target state to adjust to changing conditions.

Realizing the improvements of MPC requires constructing an accurate model of the controlled system, which can be difficult in general. However, a distributed system with simple requests (see Section 2.1) is simpler to control: by avoiding per-server SLO violations, the controller avoids global violations.

We use a model of the system that predicts SLO violations based on the workload from individual servers. An overloaded server is in danger of a violation and needs to have data moved away. Similarly, the control framework uses the model to estimate how much spare capacity is left on an underloaded server, helpful for deciding which data should be moved there. Details of our model are in Section 5.4.

3.2 Reduce Data Movement

Figure 2 demonstrates that data movement negatively impacts performance. To reduce the amount of data copied between servers, we organize data as small units (*bins*), monitor workload to these bins, and move individual bins of data. This approach is commonly used to ease load-balancing [14, 17].

Monitoring workload statistics at a granularity finer than per-server is essential for the control framework to decide which data should be moved or copied. Without this information, it would be impossible to determine the minimal amount of data that could be moved from an overloaded server to bring it back to an SLO-compliant state. The performance model can predict how much "extra room" underloaded servers have, allowing the control framework to choose where to move the data. A "best-fit" policy that keeps the servers as fully utilized

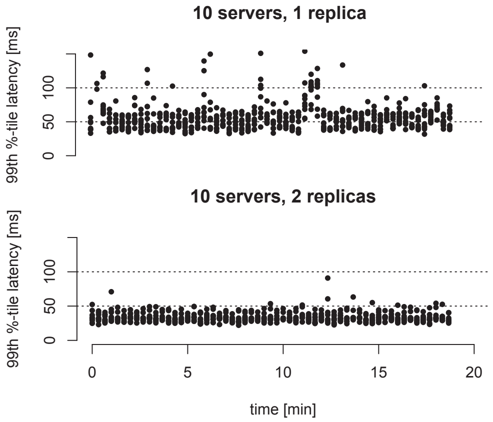


Figure 3: 99th percentile of latency over time measured during two experiments with steady workload. We kept the workload volume and number of servers the same, but changed the replication level from one data copy (top) to two (bottom). Horizontal lines representing the latencies 50 ms and 100 ms are provided for reference.

as possible is also important for scaling down leased resources, as unused servers can be released. Monitoring workload on small ranges of data give the control framework fine-grained information to move as little data as necessary to alleviate performance issues and to safely coalesce servers so they can be released.

3.3 Replication for Predictability

Distributed systems, particularly those operating in a cloud environment, typically experience environmental noise uncorrelated to a particular query or data [19]. In our benchmarks, we saw fluctuations in 99th percentile of latency over time and between different servers.

However, distributed systems also present the opportunity to use replication as a means of improving performance. In Dynamo, setting the read/write quorum parameters to be less than the total number of replicas achieves better request latency [20]. Another example is in the Google File System [21], which writes logs to different servers.

We handle performance perturbations caused by environmental noise by exploiting data replication; replication in the cloud environment is useful for performance predictability. Each request is sent to multiple replicas of the requested item and the first response is sent back to the client; this is the technique described in [20].

Figure 3 compares using one replica versus two on the same number of total servers (ten); shown is the 99th percentile of latency over time measured with steady workload. Note that the latency using replication is both smaller and more stable, even though each of these servers is doing more work than a server in the single replica scenario. It may seem that using single replicas with higher utilization would yield higher overall goodput (i.e., the amount of useful work accomplished per unit time). However, the extra work done by increasing the utilization will be in vain if those requests violate the SLO. In other words, the stringent SLO lowers the *useful* utilization of a single server.

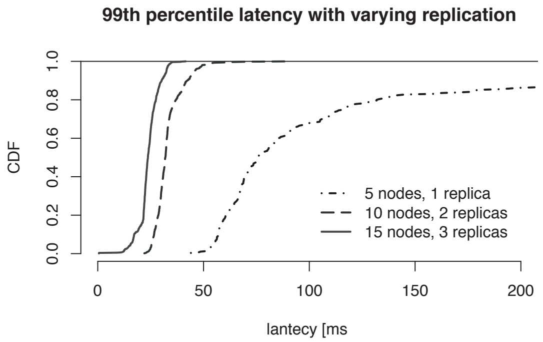


Figure 4: CDFs of 99th percentile latency measured every 20 seconds in three experiments. Each experiment yields the same goodput, however using more replicas results in lower and less variable latency.

Using more replicas yields lower variance in the 99th percentile. Figure 4 shows three Cumulative Distribution Functions (CDFs) of the 99th percentile of latency during three experiments using up to three replicas; each experiment yields the same goodput (workload to fully load five single replicas). Note the shorter tails on the distributions as the replication factor increases.

An advantage of using replication for performance is that it helps mask the effects of data movement during dynamic scaling. Thus replication is beneficial for alleviating both naturally-occurring and introduced noise.

Note that this data replication technique improves the 99th percentile latency from the perspective of the client, but does not reduce variance of the upper percentiles of latency of requests from an individual server. Therefore, the need for model-based control due to the difficulty in controlling a noisy signal remains present.

4 Related Work

Previous projects have addressed various subsets of our problem space, but to our knowledge none tackle the entire problem of the online control of the upper percentiles of latency in stateful, distributed systems.

Some work [2, 33] aims to optimize the static provisioning of a storage system before deploying to production. They search the configuration space for a cluster configuration that optimizes a specified utility function, but this optimization is done offline and performance is not considered during the re-configuration.

Other work tackles online configuration changes in storage systems, but only considers mean request latency rather than the upper percentile SLOs we consider. In

[16, 32], the authors propose a database replication policy for automatic scale up and down. In [32], they use a reactive, feed-back controller which monitors request latency and adds additional full replicas of the database. An enhancement in [16] uses a performance model to add replicas via a proactive controller. These papers additionally differ from our work in their assumption that the full dataset fits on a single server, thus they only consider adding a full replica when scaling up (instead of also partitioning).

In [25], the controller adds and removes nodes from a distributed file system, rebalancing data as servers come and go. However this work focuses more on controlling the rebalance speed rather than choosing which data to move to which servers; the work additionally does not focus on upper-percentile SLOs.

Some systems target large-scale storage servers with terabytes of data on each machine and thus cannot handle a sustained workload spike or data hotspot because the data layout cannot change on-the-fly. For example: in Everest [28], the authors propose a *write off-loading technique* that allows them to absorb short burst of writes to a large-scale storage system. Performance improvement is measured as 99th percentile of latency during the 30 minute experiments, however they do not attempt to maintain a stringent SLO over short time intervals. Sierra [35] and Rabbit [1] are power-proportional systems that alter power consumption based on workload. The approach that both papers take is to first provision the system for the peak load with multiple replicas of all data and then turn off servers when the workload decreases. Both papers evaluate the performance of the system under the power-proportional controller (Sierra uses the 99th percentile of latency), but these systems could not respond to workload spikes taller than the provisioned capacity or to unexpected hotspots that affect individual servers. SMART [38] is evaluated on a large file system that prevents it from quickly responding to unexpected spikes and does not consider upper percentiles of latency.

Most DHTs [8] are designed to withstand churn in the server population without affecting the availability and durability of the data. However, quickly adapting to changes in user workload and maintaining a stringent performance SLO during such changes are not design goals. Amazon's Dynamo [20] is an example of a DHT that provides an SLO on the 99.9th percentile of latency, but the authors mention that during a busy holiday season it took almost a day to copy data to a new server due to running the copy action slow enough to avoid performance issues; this low-priority copying would be slow to respond to unexpected spikes.

Much has been published on dynamic resource allocation for stateless systems such as Web servers or application servers [15, 36, 26, 23, 22, 34], even considering stringent performance SLOs. However, most of that work does not directly apply to stateful storage systems: the control polices for stateless systems need only vary the number of active servers because any server can handle any request. These policies do not have to consider the complexities of data movement.

Aqueduct [27] is a migration engine that moves data in a storage system while guaranteeing a performance SLO on mean request latency. It does not directly respond to workload, but could be used instead of the action scheduler in our control framework (see Section 5.6).

5 The Control Framework

This section describes the design and implementation of the control framework, incorporating the strategies outlined in Section 3. The framework uses per-server workload and the performance model to determine when a server is overloaded and thus when to copy data. It chooses *what* to copy based on workload statistics on small units of data (*bins*). Finer statistics together with the models inform *where* to copy data.

5.1 The control loop

The control framework consists of a controller, workload forecaster, and action scheduler which, together with the storage system and performance models, form a control loop (see Figure 5). These components are described in more detail in subsequent sections.

We focus on the controller, which is responsible for altering the configuration of the cluster by prescribing actions that add/remove servers and move/copy data between servers. Its decisions are based on a view of the current state given by the workload forecaster and the current data layout, in consultation with models that predict how servers will perform under particular loads. After the controller compiles a list of actions to run on the cluster, the action scheduler executes them.

Workload statistics are maintained for small ranges of

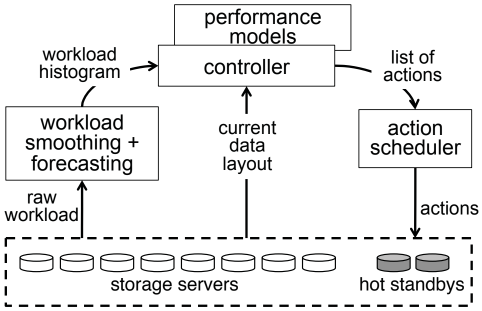


Figure 5: The control framework modules—workload forecasting, controller, performance model, and action execution—form a control loop that interacts with the storage system.

data called *bins*; each bin is about 10-100 MB of data. These bins also represent the unit of data movement. We assume a bin cannot be further partitioned and will need to be replicated if its workload exceeds the capacity of a single server. The total number of bins is a parameter of the control framework. Setting the value too low or too high has its drawbacks. With too few data bins, the controller does not have enough flexibility in terms of moving data from overloaded servers and might have to copy more data than necessary. Having too many data bins increases the load on the monitoring system and running the controller might take longer since it would have to consider more options. In practice, having on average five to ten bins per server is a good compromise.

5.2 A manipulable storage system

The SCADS [6] storage system provides an interface for dynamic scaling: it is easy to control which servers have which data, and data can be manipulated as small bins. SCADS is an eventually consistent key-value store with range partitioning. Each node can serve multiple small ranges; e.g., keys A-C, G-I. We use the `get` and `put` operators; read requests are satisfied from one or more servers, and writes are asynchronously propagated and flushed to all replicas.

SCADS provides an interface for copying and moving data between pairs of servers; replication is accomplished by copying the target data range to another server, and partitioning is the result of moving data from one server to another. The SCADS design makes low latency a top priority, thus all data is kept in memory. This characteristic has little impact on the control framework, besides simplifying the performance modeling described in Section 5.4.

5.3 Controller

Given the workload statistics in each bin, the minimal number of servers would be achieved by solving a bin-packing problem—packing the data bins into servers—an NP-complete problem. While approximate algorithms exist [37], they typically do not consider the current locations of the bins and thus could completely reshuffle the data on the servers, a costly operation. Instead, our controller uses a greedy heuristic that moves data from the overloaded servers and coalesces underloaded servers. While there are many possible controller implementations, we describe our design that leverages the solutions outlined above.

The controller executes periodically to decide how to alter the configuration of the cluster; the frequency is an implementation parameter. In each iteration, the controller prescribes actions for overloaded and underloaded servers as well as changing the number of servers. By the end of an iteration, the controller has compiled a list of actions to be run on the cluster, which are then executed by the action scheduler (see Section 5.6).

Algorithm 1 Controller iteration

```
1:  estimate workload on each server
2:  identify servers that are overloaded or underloaded
3:
4:  for all overloaded server S do
5:      while S is overloaded do
6:          determine hottest bin H on S
7:          if workload on H is too high for a single server then
8:              move and replicate H to empty servers
9:          else
10:             move H to the most-loaded underloaded server
                that can accept H without SLO violation
11:
12: for all underloaded server S do
13:     if S contains only a single bin replica then
14:         remove the bin if no longer necessary
15:     else
16:         for all bin B on S do
17:             move B to most-loaded underloaded server that
                can accept B
18:             if cannot move B then
19:                 leave it on S
20:
21: add/remove servers as necessary, as per previous actions
```

Pseudocode for the controller is shown in Algorithm 1. Using a performance model (described in the next subsection), the controller predicts which servers are underloaded or overloaded. Lines 4-10 describe the steps for fixing an overloaded server: moving bins that have too much workload for one server to dedicated servers, or moving bins to the most loaded servers that have enough capacity, a “best-fit” approach. Next, in lines 12-19, in an attempt to deallocate servers for scaling down, the controller moves bins from the least loaded loaded servers to other underloaded servers. Finally, servers are added and removed from the cluster. To simplify its reasoning about the current state of the system, the controller waits until previously scheduled copy actions complete. Long-running actions could block the controller from executing, preventing it from responding to sudden changes in workload. An action that needs to move many bins from one server to another. To avoid scheduling such actions, the controller uses a copy-duration model to estimate action duration and splits potentially long-running actions into shorter ones. For example, an action that needs to move many bins from one server to another can be split into several actions that move fewer bins between the two servers. If some of the actions do not complete within a time threshold, the controller can cancel them to reassess the current state and continue to respond to workload changes.

The controller can also maintain a user-specified number of standby servers, a form of extra capacity in addition to overprovisioning in the workload smoothing component (see Section 5.5). These standbys help the controller avoid waiting for new servers to boot up during a sudden workload spike, as they are already running the storage system software but not serving any data. Standbys are particularly useful for handling hotspots when replicas of a bin require an empty server.

The presence of a centralized component such as the controller does not necessarily mean the system isn't scalable[19]. Nevertheless, there is likely a limit to the number of decisions the controller can make per unit time for a given number of servers and/or bins. In our results, the controller inspects forty servers in a few seconds; experimenting with a larger cluster is future work. If a decision-making limit is approached, the controller may need to make decisions less frequently; this could impact the attainable SLO if the workload changes rapidly. However, with more servers, the controller has more flexibility in placing data, meaning it doesn't have to consider many servers when relocating a particular bin.

5.4 Benchmarking and modeling

The controller uses models of system performance to determine which servers are overloaded/underloaded and to guide its decisions as to which data to move where, as well as how many servers to add or remove. Recall that Model-Predictive Control requires an accurate model of the system. Instead of responding to changes in 99th percentile of request latency, our controller responds directly to changes in system workload. Therefore, the controller needs a model that accurately predicts whether a server can handle a particular workload without violating the performance SLO. Our controller also uses a model of duration of the data copy operations to create short copy actions.

One of the standard approaches to performance modeling is using analytical models based on network of queues. These models require detailed understanding of the system and often make strong assumptions about the request arrival and service time distributions. Consequently, analytical models are difficult to construct and their predictions might not match the performance of the system in production environments.

Instead, we use statistical machine learning (SML) models. As noted in the solutions above, a model-based approach allows us to use a signal other than latency in the control loop. Consequently, the controller needs an accurate model of the system on which to base its decisions. Building a model typically involves gathering training data by introducing a range of inputs into the system and observing the outcomes. In a large-scale system it becomes more difficult to construct the appropriate set of inputs [7]. Furthermore, it is more likely in a larger system to only be able to observe a subset of the component interactions that actually take place. Not having knowledge of all interactions (*unmodeled dynamics*) leads to a less accurate model.

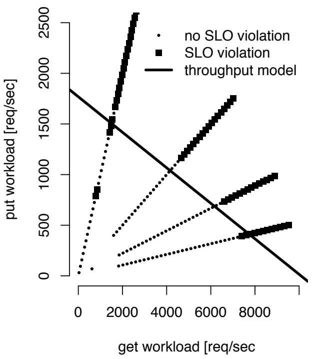


Figure 6: The training data and steady-state model for two replicas. The x- and y-axes represent the request rates of get and put operations, and the small dots and large squares represent workloads that the server can and cannot handle, respectively. The solid line crossing the four others is the boundary of the performance model. SCADS can handle workload rates to the left of this line.

Fortunately, we can leverage the simple communication pattern of SCADS requests to simplify the modeling process. Other key-value stores with similar simple requests would also be amenable to modeling. Below we describe the development and use of two models, the *steady-state* model and the *copy-duration* model. All benchmarks were run on tens of SCADS servers and workload-generating clients on Amazon's Elastic Compute Cloud (EC2) on m1.small instances.

Simple changes in workload, such as a shift in popularity of individual objects [11, 5, 9], will not affect the accuracy of these offline models as all SCADS requests are served from memory. The performance of these offline models (and thus the system) may degrade over time if new, unmodeled features are added to the application. For example, an individual request may become more expensive if it returns more data or if new types of requests are supported. The model's degradation speed would be application-specific, however these feature-change events are known to the developer and the offline models can be periodically rebuilt via benchmarking and fine-tuned in production [12].

Steady-state model: The steady-state performance model is used to predict whether a server can handle a particular workload without violating a given latency threshold. The controller uses this model to detect which servers are overloaded and to decide where data should be moved. To build this model, we benchmark SCADS under steady workload for a variety of workload mixes: read/write ratios 50/50, 80/20, 90/10 and 95/5 (these mixes are also used in [18]). We then create a linear classification model using logistic regression, based on training data from the benchmarks. The model has two covariates (features): the workload rate of get and put requests. For each workload mix, we determine the workload volume at which the latency threshold specified by the SLO would be surpassed. This workload volume separates two classes: SLO violation or no violation. Thus, for a particular workload, the model can predict whether a server with that workload would violate the SLO. Figure 6 illustrates the steady-state linear model and the training data used to generate it.

Copy-duration model: To allow the controller to estimate how long it will take to copy data between two servers, we build a model that predicts the rate of data transfer during a copy action. While the copy operation in SCADS has a parameter for specifying the number of bytes/second at which to transfer data, the actual rate is often lower because of activity on both servers involved. Our model thus predicts the *copy-rate factor*—the ratio of observed to specified copy-rate. A factor of 0.8 means that the actual copy operation is only 80% the specified rate. We use this estimate of the actual rate to compute the duration of the copy action.

To build the model, we benchmark duration of copy actions between pairs of servers operating at various workload rates. We then model the copy rate factor using linear regression; covariates are linear and quadratic in the specified rate and get and put request rates.

While our controller does not directly consider the effects of data copy on system performance during real-time decisions, we considered these effects when designing the controller and the action execution modules. Recall that Figure 2 summarizes the results of benchmarking SCADS during copy operations; performance is affected mostly on the target servers for the copy action. Also note that in both performance models network utilization and activity of other VMs are ignored. These effects are part of environmental noise described earlier, and are compensated for with replication.

5.5 Workload Monitoring and Smoothing

In addition to performance models, the controller needs to know how workload is distributed amongst the data. Workload is represented by a histogram that contains request rates for individual request types (get and put) for each bin. To minimize the impact of monitoring on performance, we sample 2% of get requests for use in our statistics (put requests are sampled at 40% because there are fewer put requests in our workload mixes). We found that using higher sampling rates did not greatly improve accuracy.

Every twenty seconds, a summary of the workload volume is generated for each bin. This creates the raw workload histogram: for each bin we have counts of the number of get and put requests to keys in that bin. To prevent the controller from reacting to small variance in workload, the raw workload is smoothed via hysteresis. As scaling up is more important than scaling down with respect to performance, we want to respond quickly to workload spikes while coalescing servers more slowly. We apply smoothing with two parameters: α_{up} and α_{down}. If the workload in a bin increases relative to the last time step's smoothed workload, we smooth that bin's workload with α_{up}; otherwise we use the α_{down} smoothing parameter. For example, in the case of increasing workload at time t we have: $smoothed_t = smoothed_{t-1} + \alpha_{up} * (raw_t - smoothed_{t-1})$.

The smoothed workload can also be amplified using an *overprovisioning* factor. Overprovisioning causes the controller to think the workload on a server is higher than it actually is. For instance, an overprovisioning factor of 0.1 would make an actual workload of w appear to the controller as $1.1w$. Thus overprovisioning creates a "safety buffer" that buys the controller more time to move data. For more discussion of tradeoffs, see Section 7.

The controller bases its decisions on an estimate of the workload at each server, determined by sampling the requests. Calculating per-bin workload in a centralized controller may prove unscalable as the number of requests to sample grows large. While we used a single server to process the requests and compute the per-bin workloads, the Chukwa monitoring system [29] could be distributed over a cluster of servers. The monitoring system could then prioritize the delivery of the monitoring data to the controller, sending updates only for bins with significant changes in workload. Another approach would have each server maintain workload information over a specified time interval. The controller could then query for the workload information when it begins its decision-making process.

5.6 Action Scheduler

On most storage systems, copying data between servers has a negative impact on performance of the interactive workload. In SCADS, the copy operation significantly affects the target server (see Figure 2), while the source server is mostly unaffected. Therefore, executing all data copy actions concurrently might overwhelm the system and reduce performance. Executing the actions sequentially would minimize the performance impact, but would be very slow.

In addition to improving steady-state performance of storage systems, replication helps smooth performance during data copy. We specify a constraint that each bin have at least one replica on a server that is not affected by data copy. The action scheduler iterates through its list of actions and schedules concurrently all actions that do not violate the constraint. When an action completes, the scheduler repeats this process with the remaining unscheduled actions.

5.7 Controller Parameters

A summary of the parameters used by the controller appear in Table 5.7, along with the values used in our experiments (in Section 6). The hysteresis parameters α_{up} and α_{down} affect how abruptly the controller will scale up and down. Reasonable values for these parameters can be chosen via simulation [13].

Controller Parameter	Value
execution period	20 seconds
α_{up}, α_{down}	0.9, 0.1
number standbys	2
overprovisioning	0.1 or 0.3
copyrate	4 MB/s

6 Experimental Results

We evaluate our control framework implementation by stress testing it with two workload profiles that represent the main scenarios where our proposed control framework could be applied. The first workload contains a spike on a single data item; as shown in [11], web applications typically experience hotspots on a small fraction of the data. Unexpected workload spikes with data hotspots are difficult to handle in stateful systems because the location of the hotspot is unknown before the spike. Therefore, statically overprovisioning for such spikes would be expensive. Managing and monitoring small data ranges is especially important for dealing with these hotspots, particularly when quick replication is needed. The second workload exhibits a diurnal workload pattern: workload volume increases during the day and decreases at night; this profile demonstrates the effectiveness of both scale-up and scale-down.

For the hotspot workload, we observe how well the control framework is able to react to a sudden increase in workload volume, as well as how quickly performance stabilizes. We also look at the performance impact during this transition period. Note, however, that any system will likely have some visible impact for sufficiently strict characteristics of the spike (i.e., how rapidly it arrives and how much extra workload there is). The diurnal workload additionally exercises the control framework's ability to both scale up and down. Finally, we discuss some of the tradeoffs of SLO parameters and cost of leased resources, as well as potential savings to be gained by scaling up and down.

6.1 Experiment setup

Experiments were run using Amazon's Elastic Compute Cloud (EC2). We ran SCADS servers on m1.small instances using 800 MB of the available RAM as each storage server's in-memory cache. We gained an understanding of the variance present in this environment by benchmarking SCADS' performance both in the absence and presence of data movement, see Section 5.4. As described in Section 2, latency variance occurs in the upper quantiles even in the absence of data movement. Therefore we maintain at least two copies of each data item, using the replication strategy described earlier: each `get` request is sent to both replicas and we count the faster response as its latency. We do not consider the latency of `put` requests, as the work described in this paper is targeted towards OLTP-type applications similar to those described by [18], in which read requests dominate writes. Furthermore, evaluating latency for write requests isn't applicable in an eventually consistent system, such as SCADS. More appropriate would be an SLO on data staleness, a subject for future work.

Workload is generated by a separate set of machines, also m1.small instances on EC2. These experiments use sixty workload-generating instances and twenty server instances. The control framework runs on one m1.xlarge instance. The controller uses a 100 ms SLO threshold on latency for `get` requests, and in the description of each experiment we discuss the other two parameters of the SLO: the percentile at which to evaluate the threshold, and the interval over which to assess violations. Table 1 summarizes the parameter values used in the two experiments. To avoid running an experiment for an entire day, we execute it in a shorter time. We control the length of the boot-up time in the experiment by leasing all the virtual machines needed before the experiment begins and simply adding a delay before a "new" server can be used. This technique allows us to replay the Ebates.com work-

Parameter	Hotspot	Diurnal
server boot-up time	3 minutes	15 seconds
server charge interval	60 minutes	5 minutes
server capacity	800 MB	66.7 MB
size of 1 key-value	256 B	256 B
total number of keys	4.8 million	400,000
minimum # of replicas	2	2
total data size	2.2 GB	196 MB
read/write ratio	95/5	95/5

Table 1: Various experiment parameters for the hotspot and diurnal workload experiments. We replay the diurnal workload with a speed-up factor of 12 and thus also reduce the server boot-up and charge intervals and the data size by a factor of 12.

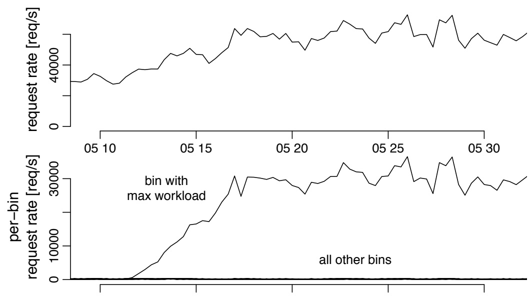


Figure 7: Workload over time in the Hotspot experiment. Top row: aggregate request rate during the spike doubled between 5:12 and 5:17. Bottom row: request rate for each of the 200 data bins; the rate for the hot bin increased to approximately 30,000 reqs/sec.

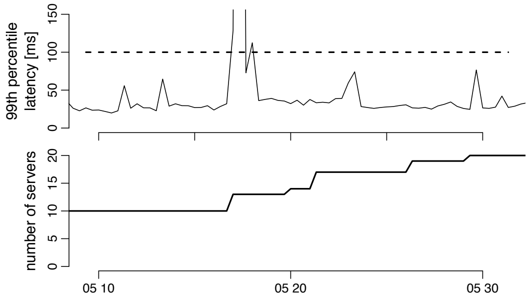


Figure 8: Performance and resources in the Hotspot experiment. Top row: 99th percentile of latency along with the 100 ms threshold (dashed line). Bottom row: number of servers over time. The controller keeps up with the spike for the first few minutes, then latency increases above the threshold, but the system quickly recovers.

Interval	Max percentile
5 minutes	98
1 minute	95
20 seconds	80

Table 2: The maximum percentile without SLO violations for each interval in the Hotspot experiment. Notice that we can support higher latency percentiles for longer time intervals.

load trace [10] 12x faster: replaying twenty-four hours of the trace in two hours. To retain the proportionality of the other time-related parameters, we scale down by 12x the data size, server cost interval, boot up time, and server release time. The data size is scaled down because we can't speed up the copy rate higher than the network bandwidth on m1.small instances allows. Additionally, the total data size is limited by the maximum storage on the number of servers when the cluster is scaled down. As SCADS keeps its data in memory, server capacity is limited by available memory on the m1.small instance.

6.2 Hotspot

We create a synthetic spike workload based on the statistics of a spike experienced by CNN.com after the September 11 attacks [24]. The workload increased by an order of magnitude in 15 minutes, which corresponds to about 100% increase in 5 minutes. We simulate this workload by using a flat, one-hour long period of the Ebates.com trace [10] to which we add a workload spike with a single hotspot. During a five minute period, the aggregate workload volume increases linearly by a factor of two, but all the additional workload is directed at a single key in the system. Figure 7 depicts the aggregate workload and the per-bin workload over time. Notice that when the spike occurs, the workload in the hot bin greatly exceeds that in all other bins.

Our controller dynamically creates eight additional replicas of this hot data bin to handle the spike. Figure 8 shows the performance (99th percentile latency) and the number of servers over time. The workload spike impacts performance for a brief period. However, the controller quickly begins replicating the hot data bin. It first uses the two standbys, then requests additional servers. Performance stabilizes in less than three minutes.

It is relatively easy for our control framework to react to spikes like this because only a very small fraction of the data has to be replicated. We can thus handle a spike with data hotspots with resources proportional to the magnitude of the spike, not proportional to the size of the full dataset or the number of servers.

The performance impact when the spike first arrives is brief, but may result in an SLO violation, depending how the SLO is specified. The SLO is parameterized by the latency threshold, latency percentile, and duration of the SLO interval. Fixing the latency threshold at 100 ms, in Table 6.2 we show how varying the interval affects the maximum percentile under which no violations occurred.

In general, SLOs specified over a longer time interval are easier to maintain despite drastic workload changes; this experiment has one five-minute violation. Similarly, an SLO with a lower percentile will have fewer violations than a higher one. In this experiment, there are zero violations over a twenty-second window when looking at the 80th percentile of latency, but extending the interval to five minutes can yield the 98th percentile.

The cost tradeoff between SLO violations and leased resources depends in part on the cost of a violation. Whether a violation costs more than leasing enough servers to overprovision the system to satisfy a hotspot on *any* data item will be application-specific. Dynamic scaling, however, has the advantage of not having to estimate the magnitude of unexpected spikes.

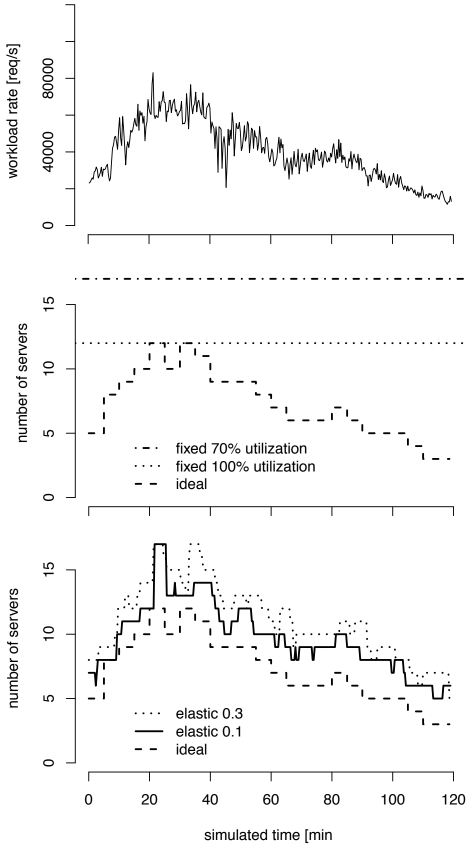


Figure 9: Top: Diurnal workload pattern. Middle: number of servers assuming the ideal server allocation and two fixed allocations during the diurnal workload experiment. Bottom: ideal server allocation and two elastic allocations using our control framework.

6.3 Ebates.com diurnal workload trace

The diurnal workload profile is derived from a trace from Ebates.com [10]; we use the trace's aggregate workload pattern; data accesses follow a constant zipfian distribution. This profile shows the control framework's effectiveness in scaling both up and down as the workload volume on all data items fluctuates. We replay twenty-four hours of the trace in two hours, a 12x speedup.

We experiment using two overprovisioning parameters (see Section 5.5 on workload smoothing). With 0.3 overprovisioning, the smoothed workload is multiplied by a factor of 1.3. With more headroom, the system can better absorb small spikes in the workload. Using 0.1 overprovisioning has less headroom, thus higher savings at the cost of worse performance.

We compare the results of our experiments with the ideal resource allocation and two fixed allocation calculations. In the ideal allocation, we assume that we know the workload at each time step throughout the experiment and compute the minimum number of servers we would need to support this workload for each 5-minute interval (the scaled-down server cost interval). The ideal allocation assumes that moving data is instantaneous and has no effect on performance, and provides the lower bound on the number of compute resources required to handle this workload without SLO violations.

The fixed-100% and fixed-70% allocations use a constant number of servers throughout the experiment. Fixed-100% assumes the workload's peak value is known a priori, and computes the number of servers based on that value and the maximum throughput of each server (7000 requests per second, see Section 5.4). The number of servers used in the fixed-100% allocation equals the maximum number of servers used by the ideal allocation. Fixed-70% is calculated similarly to the fixed-100%, but restricts the servers' utilization to 70% of their potential throughput (i.e., $7,000 * 0.7 = 4,900$ requests per second). Fixed-100% is the ideal fixed allocation, but in practice datacenter operators often add more headroom to absorb unexpected spikes.

Figure 9 shows the workload profile and the number of server units used by the different allocation policies: ideal, fixed-100%, fixed-70%, and our elastic policy with overprovisioning of 0.3 and 0.1. A server unit corresponds to one server being used for one charge interval, thus fewer server units used translates to monetary cost savings. The policy with 0.1 overprovisioning achieves savings of 16% and 41% compared to the fixed-100% and fixed-70% allocations, respectively.

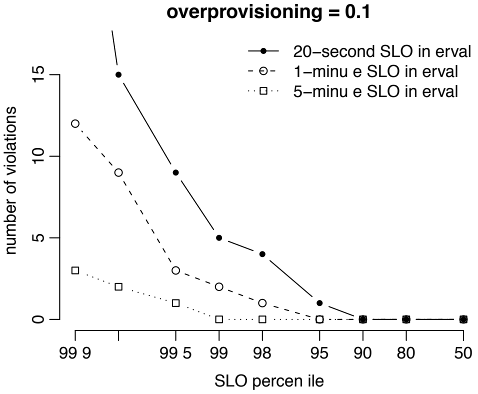
The ideal resource allocation uses 175 servers units, while using overprovisioning of 0.1 uses 241 server units. However, recall that our controller maintains two empty standby servers to quickly respond to data hotspots that require replication. The actual number of server units used for serving data is thus 191 which is within 10% of the ideal allocation[3].

Performance and SLO violations are summarized in Figure 10. Note that it is more difficult to maintain SLOs with shorter time intervals and higher percentiles.

7 Discussion

The experiments demonstrate the control framework's effectiveness in scaling both up and down for typical workload profiles that exhibit fluctuating workload patterns. Having the same mechanism work well in scenarios with rapidly appearing hotspots as well as more gradual variations is advantageous because application developers won't need to decide a priori what type of growth to prepare for: the same control framework can dynamically scale as needed in either case. For operators who still prefer to maintain a fixed allocation for non-spike, peak traffic, say on their own hardware, there is still potential to utilize the control framework for surge computing in the cloud. A temporary spike could be sat-

[3] The experiment has a total of 25 5-minute server-charging intervals which yields 50 server units used by the standbys.



Interval	Max percentile	
	0.3 overprovision	0.1 overprovision
5 minutes	99.5	99
1 minute	99	95
20 seconds	95	90

Figure 10: Top: Number of SLO violations during the 0.1 overprovisioning diurnal experiment, for different values of the SLO percentile. The three lines in the graph correspond to the three intervals over which to evaluate the SLO: 5 minutes, 1 minute, and 20 seconds. Bottom: summary of SLO violations and maximum latency percentile supported with no SLO violations during the diurnal workload with two different overprovisioning parameters.

isfied with leased resources from the cloud, which would be relinquished once the spike subsides.

There are cost implications in setting some of the control framework's parameters to manage "extra capacity," namely the number of standbys and the overprovisioning factor. Both these techniques result in higher server costs, either due to maintaining booted empty servers for standbys or underutilization of active servers in the case of overprovisioning. Standby servers are particularly helpful for dealing with workload spikes which necessitate replication, as empty servers are waiting and ready to receive data. Overprovisioning is better for workload profiles like a diurnal pattern in which all data items more slowly experience increased access rates; this headroom allows the control framework more time to shuffle data around without overloading the servers. Reducing the number of standbys and/or the overprovisioning factor can yield cost savings, with the associated risk of SLO violations if scaling up is not performed rapidly enough.

We presented results of our controller using replication to both smooth variance and lessen the effects of data movement. To see that the controller remains robust to the variance in the environment without replication, we performed the same two experiments using only a single copy of each data item. While SCADS still scales effectively, the variance limits the attainable SLO percentile. For example, in the hotspot workload, the 5-minute, 1-minute, and 20-second attainable percentiles were 95, 80, and 80, respectively, compared to 98, 95, and 80 when using replication. The replication factor thus offers a tradeoff between performance/robustness and the cost of running the system. Note, however, that a different environment than EC2, like dedicated hardware, may have less variance and thus may achieve the desired SLO without replication.

The ability to control these performance tradeoffs is an advantage of running the SCADS key-value store on EC2 rather than simply using S3 for data storage. In general, S3 is optimized for larger files and has nontrivial overhead per HTTP request. S3 also does not offer a SLO on latency, while SCADS offers a developer-specified SLO. Data replication factor and data location are not tunable with S3, which would make maintaining a particular SLO difficult. More fundamentally, S3 does not provide the API that SCADS on EC2 does. SCADS supports features like `TestAndSet()` and various methods on ranges of keys; this enables a higher level query language on top. Additionally, the SCADS client library supports read/write quorums for trading off performance and consistency, this would also be meaningless without being able to control the replication factor.

8 Future Work

Future work includes incorporating resource heterogeneity in the control framework, as well as designing a framework simulator for performing what-if analysis. Cloud providers typically offer a variety of resources at different cost, e.g., paying more per hour for a server with more CPU or disk capacity. By modeling performance of different server types, we could include in the control framework decisions about which type of server to use. Additionally, we hope to use the performance models in a control framework simulator that emulates the behavior of real servers. The simulator could be used for assessing the performance-cost tradeoff for unseen workloads; developers could create synthetic workloads using the features described in [11].

9 Conclusion

The elasticity of the cloud provides an opportunity for dynamic resource allocation, scaling up when workload increases and scaling down to save money. To date, this opportunity has been exploited primarily by stateless services, in which simply adding and removing servers is sufficient to track workload variation. Our goal was to design a control framework that could automatically scale a *stateful* key-value store in the cloud while complying with a stringent performance SLO in which a very high percentile of requests (typically 99%) must meet a specific latency bound. As described in Section 2, meeting such a stringent SLO is challenging both because of high variance in the tail of the request latency distribution and because of the need to copy data in addition to adding and removing servers. Our solution avoids trying to control for such a noisy latency signal, instead using a model-based approach that maps workload to latency. This model, combined with fine-grained workload statistics, allows the framework to move only necessary data to alleviate performance issues while keeping the amount of leased resources needed to satisfy the current workload. In the event of an unexpected hotspot, replicas are added proportional to the magnitude of the spike, not the total number of servers. For workload that exhibits a diurnal pattern, the framework easily scales both up and down as the workload fluctuates. In the midst of this dynamic scaling, we use replication to mask both inherent environmental noise and the performance perturbations introduced by data movement. We anticipate that this work provides a useful starting point for allowing large-scale storage systems to take advantage of the elasticity of cloud computing.

10 Acknowledgements

We would like to thank our fellow students and faculty in the RAD lab for their ongoing thoughtful advice throughout this project. We additionally thank Kim Keeton, Tim Kraska, Ari Rabkin, Eno Thereska, and John Wilkes for their feedback on this paper.

This research is supported in part by a National Science Foundation graduate fellowship, and gifts from Sun Microsystems, Google, Microsoft, Amazon Web Services, Cisco Systems, Cloudera, eBay, Facebook, Fujitsu, Hewlett-Packard, Intel, Network Appliance, SAP, VMWare and Yahoo! and by matching funds from the State of California's MICRO program (grants 06-152, 07-010, 06-148, 07-012, 06-146, 07-009, 06-147, 07-013, 06-149, 06-150, and 07-008), the National Science Foundation (grant #CNS-0509559), and the University of California Industry/University Cooperative Research Program (UC Discovery) grant COM07-10240.

References

[1] H. Amur, J. Cipar, V. Gupta, G. R. Ganger, M. A. Kozuch, and K. Schwan. Robust and Flexible Power-Proportional Storage. In *SoCC: ACM Symposium on Cloud Computing*, 2010.

[2] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: Running Circles Around Storage Administration. In *FAST: Conference on File and Storage Technologies*, 2002.

[3] Apache. Cassandra. incubator.apache.org/cassandra, 2010.

[4] Apache. HBase. hadoop.apache.org/hbase, 2010.

[5] M. Arlitt and T. Jin. Workload Characterization of the 1998 World Cup Web Site. Technical Report HPL-1999-35R1, HP Labs, 1999.

[6] M. Armbrust, A. Fox, D. Patterson, N. Lanham, H. Oh, B. Trushkowsky, and J. Trutna. SCADS: Scale-independent Storage for Social Computing Applications. In *Conference on Innovative Data Systems Research (CIDR)*, 2009.

[7] K. J. Åström. *Introduction to Stochastic Control Theory*. Academic Press, 1970.

[8] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking Up Data in P2P Systems. *Communications of the ACM*, 46(2), February 2003.

[9] P. Barford and M. Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Proceedings of the ACM SIGMETRICS Joint International Conference*, 1998.

[10] P. Bodík et al. Combining Visualization and Statistical Analysis to Improve Operator Confidence and Efficiency for Failure Detection and Localization. In *International Conference on Autonomic Computing (ICAC)*, 2005.

[11] P. Bodík, A. Fox, M. J. Franklin, M. I. Jordan, and D. Patterson. Characterizing, Modeling, and Generating Workload Spikes for Stateful Services. In *SoCC: ACM Symposium on Cloud Computing*, 2010.

[12] P. Bodík, R. Griffith, C. Sutton, A. Fox, M. I. Jordan, and D. A. Patterson. Automatic Exploration of Datacenter Performance Regimes. In *Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds*, 2009.

[13] P. Bodík, R. Griffith, C. Sutton, A. Fox, M. I. Jordan, and D. A. Patterson. Statistical Machine Learning Makes Automatic Control Practical for Internet Datacenters. In *Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2009.

[14] F. Chang et al. Bigtable: A Distributed Storage System for Structured Data. In *OSDI: Symposium on Operating Systems Design and Implementation*, 2006.

[15] J. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle. Managing Energy and Server Resources in Hosting Centers. In *Symposium on Operating Systems Principles (SOSP)*, 2001.

[16] J. Chen, G. Soundararajan, and C. Amza. Autonomic Provisioning of Backend Databases in Dynamic Content Web Servers. In *International Conference on Autonomic Computing (ICAC)*, 2006.

[17] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2008.

[18] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC: ACM Symposium on Cloud Computing*, 2010.

[19] J. Dean. Evolution and Future Directions of Large-scale Storage and Computation Systems at Google. In *SoCC: ACM Symposium on Cloud Computing*, 2010.

[20] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Symposium on Operating Systems Principles (SOSP)*, 2007.

[21] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Symposium on Operating Systems Principles (SOSP)*, 2003.

[22] J. L. Hellerstein, V. Morrison, and E. Eilebrecht. Optimizing Concurrency Levels in the .NET ThreadPool: A Case Study of Controller Design and Implementation. In *Workshop on Feedback Control Implementation and Design in Computing Systems and Networks*, 2008.

[23] D. Kusic et al. Power and Performance Management of Virtualized Computing Environments Via Lookahead Control. In *International Conference on Autonomic Computing (ICAC)*, 2008.

[24] W. LeFebvre. CNN.com: Facing a world crisis. www.tcsa.org/lisa2001/cnn.txt, 2001.

[25] H. C. Lim, S. Babu, and J. S. Chase. Automated Control for Elastic Storage. In *International Conference on Autonomic Computing (ICAC)*, 2010.

[26] X. Liu, J. Heo, L. Sha, and X. Zhu. Adaptive Control of Multi-Tiered Web Applications Using Queueing Predictor. *Network Operations and Management Symposium*, April 2006.

[27] C. Lu, G. A. Alvarez, and J. Wilkes. Aqueduct: Online Data Migration with Performance Guarantees. In *FAST: Conference on File and Storage Technologies*, 2002.

[28] D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. I. T. Rowstron. Everest: Scaling Down Peak Loads Through I/O Off-Loading. In *OSDI: Symposium on Operating Systems Design and Implementation*, 2008.

[29] A. Rabkin and R. H. Katz. Chukwa: A System for Reliable Large-scale Log Collection. Master's thesis, EECS Department, University of California, Berkeley, Mar 2010.

[30] J. A. Rossiter. *Model Based Predictive Control: A Practical Approach*. CRC Press, 2003.

[31] J. Rothschild. High Performance at Massive Scale - Lessons Learned at Facebook. http://cns.ucsd.edu/lecturearchive09.shtml#Roth, October 2009.

[32] G. Soundararajan, C. Amza, and A. Goel. Database Replication Policies for Dynamic Content Applications. In *EuroSys: ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2006.

[33] J. D. Strunk, E. Thereska, C. Faloutsos, and G. R. Ganger. Using Utility to Provision Storage Systems. In *FAST: Conference on File and Storage Technologies*, 2008.

[34] G. Tesauro, N. Jong, R. Das, and M. Bennani. A Hybrid Reinforcement Learning Aproach to Autonomic Resource Allocation. In *International Conference on Autonomic Computing (ICAC)*, 2006.

[35] E. Thereska, A. Donnelly, and D. Narayanan. Sierra: A Power-Proportional, Distributed storage System. Technical Report MSR-TR-2009-153, Microsoft, 2009.

[36] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal. Dynamic Provisioning of Multi-tier Internet Applications. In *International Conference on Autonomic Computing (ICAC)*, 2005.

[37] V. V. Vazirani. *Approximation Algorithms*. Springer, 2003.

[38] L. Yin, S. Uttamchandani, M. Korupolu, K. Voruganti, and R. Katz. SMART: An Integrated Multi-Action Advisor for Storage Systems. In *USENIX Annual Technical Conference*, 2006.

Scale and Concurrency of GIGA+:
File System Directories with Millions of Files

Swapnil Patil and Garth Gibson
Carnegie Mellon University
{firstname.lastname @ cs.cmu.edu}

Abstract – We examine the problem of scalable file system directories, motivated by data-intensive applications requiring millions to billions of small files to be ingested in a single directory at rates of hundreds of thousands of file creates every second. We introduce a POSIX-compliant scalable directory design, GIGA+, that distributes directory entries over a cluster of server nodes. For scalability, each server makes only local, independent decisions about migration for load balancing. GIGA+ uses two internal implementation tenets, asynchrony and eventual consistency, to: (1) partition an index among all servers without synchronization or serialization, and (2) gracefully tolerate stale index state at the clients. Applications, however, are provided traditional strong synchronous consistency semantics. We have built and demonstrated that the GIGA+ approach scales better than existing distributed directory implementations, delivers a sustained throughput of more than 98,000 file creates per second on a 32-server cluster, and balances load more efficiently than consistent hashing.

1 Introduction

Modern file systems deliver scalable performance for large files, but not for large numbers of files [18, 67]. In particular, they lack scalable support for ingesting millions to billions of small files in a single directory - a growing use case for data-intensive applications [18, 44, 50]. We present a file system directory service, GIGA+, that uses highly concurrent and decentralized hash-based indexing, and that scales to store at least millions of files in a single POSIX-compliant directory and sustain hundreds of thousands of creates insertions per second.

The key feature of the GIGA+ approach is to enable higher concurrency for index mutations (particularly creates) by eliminating system-wide serialization and synchronization. GIGA+ realizes this principle by aggressively distributing large, mutating directories over a cluster of server nodes, by disabling directory entry caching in clients, and by allowing each node to migrate, without notification or synchronization, portions of the directory for load balancing. Like traditional hash-based distributed indices [17, 36, 52], GIGA+ incrementally hashes a directory into a growing number of partitions. However, GIGA+ tries harder to eliminate synchronization and prohibits migration if load balancing is unlikely to be improved.

Clients do not cache directory entries; they cache only the directory index. This cached index can have stale pointers to servers that no longer manage specific ranges in the space of the hashed directory entries (filenames). Clients using stale index values to target an incorrect server have their cached index corrected by the incorrectly targeted server. Stale client indices are aggressively improved by transmitting the history of splits of all partitions known to a server. Even the addition of new servers is supported with minimal migration of directory entries and delayed notification to clients. In addition, because 99.99% of the directories have less than 8,000 entries [4, 14], GIGA+ represents small directories in one partition so most directories will be essentially like traditional directories.

Since modern cluster file systems have support for data striping and failure recovery, our goal is not to compete with all feature of these systems, but to offer additional technology to support high rates of mutation of many small files.[1] We have built a skeleton cluster file system with GIGA+ directories that layers on existing lower layer file systems using FUSE [19]. Unlike the current trend of using special purpose storage systems with custom interfaces and semantics [6, 20, 54], GIGA+ directories use the traditional UNIX VFS interface and provide POSIX-like semantics to support unmodified applications.

Our evaluation demonstrates that GIGA+ directories scale linearly on a cluster of 32 servers and deliver a throughput of more than 98,000 file creates per second – outscaling the Ceph file system [63] and the HBase distributed key-value store [26], and exceeding petascale scalability requirements [44]. GIGA+ indexing also achieves effective load balancing with one to two orders of magnitude less re-partitioning than if it was based on consistent hashing [30, 58].

In the rest of the paper, we present the motivating use cases and related work in Section 2, the GIGA+ indexing design and implementation in Sections 3-4, the evaluation results in Section 5, and conclusion in Section 6.

[1] OrangeFS is currently integrating a GIGA+ based distributed directory implementation into a system based on PVFS [2, 45].

2 Motivation and Background

Over the last two decades, research in large file systems was driven by application workloads that emphasized access to very *large files*. Most cluster file systems provide scalable file I/O bandwidth by enabling parallel access using techniques such as data striping [20, 21, 25], object-based architectures [21, 39, 63, 66] and distributed locking [52, 60, 63]. Few file systems scale metadata performance by using a coarse-grained distribution of metadata over multiple servers [16, 46, 52, 63]. But most file systems cannot scale access to a *large number of files*, much less efficiently support concurrent creation of millions to billions of files in a single directory. This section summarizes the technology trends calling for scalable directories and how current file systems are ill-suited to satisfy this call.

2.1 Motivation

In today's supercomputers, the most important I/O workload is checkpoint-restart, where many parallel applications running on, for instance, ORNL's CrayXT5 cluster (with 18,688 nodes of twelve processors each) periodically write application state into a file per process, all stored in one directory [7, 61]. Applications that do this per-process checkpointing are sensitive to long file creation delays because of the generally slow file creation rate, especially in one directory, in today's file systems [7]. Today's requirement for 40,000 file creates per second in a single directory [44] will become much bigger in the impending Exascale-era, when applications may run on clusters with up to billions of CPU cores [31].

Supercomputing checkpoint-restart, although important, might not be a sufficient reason for overhauling the current file system directory implementations. Yet there are diverse applications, such as gene sequencing, image processing [62], phone logs for accounting and billing, and photo storage [6], that essentially want to store an unbounded number of files that are logically part of one directory. Although these applications are often using the file system as a fast, lightweight "key-value store", replacing the underlying file system with a database is an oft-rejected option because it is undesirable to port existing code to use a new API (like SQL) and because traditional databases do not provide the scalability of cluster file systems running on thousands of nodes [3, 5, 53, 59].

Authors of applications seeking lightweight stores for lots of small data can either rewrite applications to avoid large directories or rely on underlying file systems to improve support for large directories. Numerous applications, including browsers and web caches, use the former approach where the application manages a large logical directory by creating many small, intermediate sub-directories with files hashed into one of these sub-directories. This paper chose the latter approach because users prefer this solution. Separating large directory management from applications has two advantages. First, developers do not need to re-implement large directory management for every application (and can avoid writing and debugging complex code). Second, an application-agnostic large directory subsystem can make more informed decisions about dynamic aspects of a large directory implementation, such as load-adaptive partitioning and growth rate specific migration scheduling.

Unfortunately most file system directories do not currently provide the desired scalability: popular local file systems are still being designed to handle little more than tens of thousands of files in each directory [43, 57, 68] and even distributed file systems that run on the largest clusters, including HDFS [54], GoogleFS [20], PanFS [66] and PVFS [46], are limited by the speed of the single metadata server that manages an entire directory. In fact, because GoogleFS scaled up to only about 50 million files, the next version, ColossusFS, will use BigTable [12] to provide a distributed file system metadata service [18].

Although there are file systems that distribute the directory tree over different servers, such as Farsite [16] and PVFS [46], to our knowledge, only three file systems now (or soon will) distribute single large directories: IBM's GPFS [52], Oracle's Lustre [38], and UCSC's Ceph [63].

2.2 Related work

GIGA+ has been influenced by the scalability and concurrency limitations of several distributed indices and their implementations.

GPFS: GPFS is a shared-disk file system that uses a distributed implementation of Fagin's extendible hashing for its directories [17, 52]. Fagin's extendible hashing dynamically doubles the size of the hash-table pointing pairs of links to the original bucket and expanding only the overflowing bucket (by restricting implementations to a specific family of hash functions) [17]. It has a two-level hierarchy: buckets (to store the directory entries) and a table of pointers (to the buckets). GPFS represents each bucket as a disk block and the pointer table as the block pointers in the directory's i-node. When the directory grows in size, GPFS allocates new blocks, moves some of the directory entries from the overflowing block into the new block and updates the block pointers in the i-node.

GPFS employs its client cache consistency and distributed locking mechanism to enable concurrent access to a shared directory [52]. Concurrent readers can cache the directory blocks using shared reader locks, which enables high performance for read-intensive workloads. Concurrent writers, however, need to acquire write locks from the lock manager before updating the directory blocks stored on the shared disk storage. When releasing (or acquiring) locks, GPFS versions before 3.2.1 force the directory block to be flushed to disk (or read back from disk) inducing high I/O overhead. Newer releases of GPFS have modified the cache consistency protocol to send the directory insert requests directly to the current lock holder, instead of getting the block through the shared disk subsystem [1, 22, 27]. Still GPFS continues to synchronously write the directory's i-node (i.e., the mapping state) invalidating client caches to provide strong consistency guarantees [1]. In contrast, GIGA+ allows the mapping state to be stale at the client and never be shared between servers, thus seeking even more scalability.

Lustre and Ceph: Lustre's proposed clustered metadata service splits a directory using a hash of the directory entries only once over all available metadata servers when it exceeds a threshold size [37, 38]. The effectiveness of this "split once and for all" scheme depends on the eventual directory size and does not respond to dynamic increases in the number of servers. Ceph is another object-based cluster file system that uses dynamic sub-tree partitioning of the namespace and hashes individual directories when they get too big or experience too many accesses [63, 64]. Compared to Lustre and Ceph, GIGA+ splits a directory incrementally as a function of size, i.e., a small directory may be distributed over fewer servers than a larger one. Furthermore, GIGA+ facilitates dynamic server addition achieving balanced server load with minimal migration.

Linear hashing and LH:* Linear hashing grows a hash table by splitting its hash buckets in a linear order using a pointer to the *next* bucket to split [34]. Its distributed variant, called LH* [35], stores buckets on multiple servers and uses a central split coordinator that advances permission to split a partition to the next server. An attractive property of LH* is that it does not update a client's mapping state synchronously after every new split.

GIGA+ differs from LH* in several ways. To maintain consistency of the split pointer (at the coordinator), LH* splits only one bucket at a time [35, 36]; GIGA+ allows any server to split a bucket at any time without any coordination. LH* offers a complex partition pre-split optimization for higher concurrency [36], but it causes LH* clients to continuously incur some addressing errors even after the index stops growing; GIGA+ chose to minimize (and stop) addressing errors at the cost of more client state.

Consistent hashing: Consistent hashing divides the hash-space into randomly sized ranges distributed over server nodes [30, 58]. Consistent hashing is efficient at managing membership changes because server changes split or join hash-ranges of adjacent servers only, making it popular for wide-area peer-to-peer storage systems that have high rates of membership churn [13, 42, 48, 51]. Cluster systems, even though they have much lower churn than Internet-wide systems, have also used consistent hashing for data partitioning [15, 32], but have faced interesting challenges.

As observed in Amazon's Dynamo, consistent hashing's data distribution has a high load variance, even after using "virtual servers" to map multiple randomly sized hash-ranges to each node [15]. GIGA+ uses threshold-based binary splitting that provides better load distribution even for small clusters. Furthermore, consistent hashing systems assume that every data-set needs to be distributed over many nodes to begin with, i.e., they do not have support for incrementally growing data-sets that are mostly small – an important property of file system directories.

Other work: DDS [24] and Boxwood [40] also used scalable data-structures for storage infrastructure. While both GIGA+ and DDS use hash tables, GIGA+'s focus is on directories, unlike DDS's general cluster abstractions, with an emphasis on indexing that uses inconsistency at the clients; a non-goal for DDS [24]. Boxwood proposed primitives to simplify storage system development, and used B-link trees for storage layouts [40].

3 GIGA+ Indexing Design

3.1 Assumptions

GIGA+ is intended to be integrated into a modern cluster file system like PVFS, PanFS, GoogleFS, HDFS etc. All these scalable file systems have good fault tolerance usually including a consensus protocol for node membership and global configuration [9, 29, 65]. GIGA+ is not designed to replace membership or fault tolerance; it avoids this where possible and employs them where needed.

GIGA+ design is also guided by several assumptions about its use cases. First, most file system directories start small and remain small; studies of large file systems have found that 99.99% of the directories contain fewer than 8,000 files [4, 14]. Since only a few directories grow to really large sizes, GIGA+ is designed for incremental growth, that is, an empty or a small directory is initially stored on one server and is partitioned over an increasing number of servers as it grows in size. Perhaps most beneficially, incremental growth in GIGA+ handles adding servers gracefully. This allows GIGA+ to avoid degrading small directory performance; striping small directories across multiple servers will lead to inefficient resource utilization, particularly for directory scans (using `readdir()`) that will incur disk-seek latency on all servers only to read tiny partitions.

Second, because GIGA+ is targeting concurrently shared directories with up to billions of files, caching such directories at each client is impractical: the directories are too large and the rate of change too high. GIGA+ clients do not cache directories and send all directory operations to a server. Directory caching only for small rarely changing directories is an obvious extension employed, for example, by PanFS [66], that we have not yet implemented.

Finally, our goal in this research is to complement existing cluster file systems and support unmodified applications. So GIGA+ directories provide the strong consistency for directory entries and files that most POSIX-like file systems provide, i.e., once a client creates a file in a directory all other clients can access the file. This strong

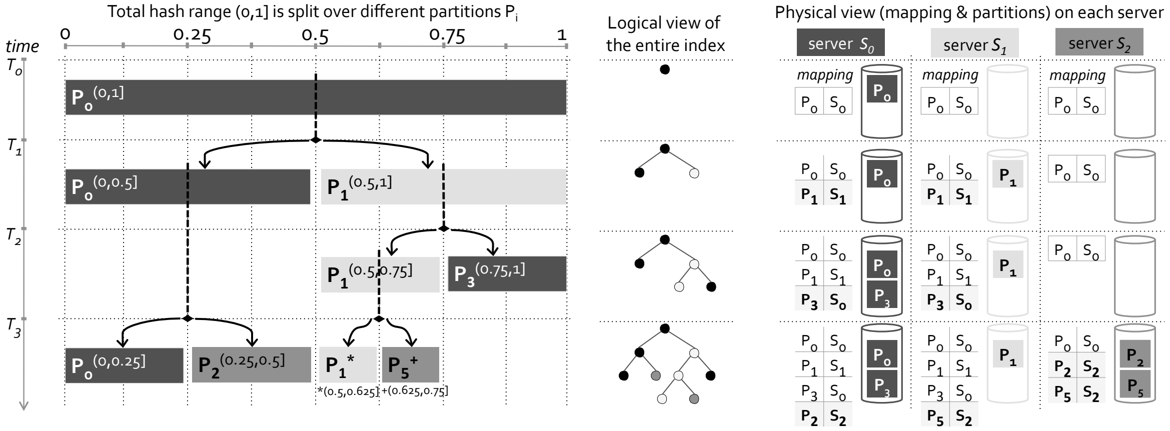


Figure 1 – Concurrent and unsynchronized data partitioning in GIGA+. *The hash-space* $(0,1]$ *is divided into multiple partitions* (P_i) *that are distributed over many servers (different shades of gray). Each server has a local, partial view of the entire index and can independently split its partitions without global co-ordination. In addition to enabling highly concurrent growth, an index starts small (on one server) and scales out incrementally.*

consistency API differentiates GIGA+ from "relaxed" consistency provided by newer storage systems including NoSQL systems like Cassandra [32] and Dynamo [15].

3.2 Unsynchronized data partitioning

GIGA+ uses hash-based indexing to incrementally divide each directory into multiple partitions that are distributed over multiple servers. Each filename (contained in a directory entry) is hashed and then mapped to a partition using an index. Our implementation uses the cryptographic MD5 hash function but is not specific to it. GIGA+ relies only on one property of the selected hash function: for any distribution of unique filenames, the hash values of these filenames must be uniformly distributed in the hash space [49]. This is the core mechanism that GIGA+ uses for load balancing.

Figure 1 shows how GIGA+ indexing grows incrementally. In this example, a directory is to be spread over three servers $\{S_0, S_1, S_2\}$ in three shades of gray color. $P_i^{(x,y]}$ denotes the hash-space range $(x,y]$ held by a partition with the unique identifier i.[2] GIGA+ uses the identifier i to map P_i to an appropriate server S_i using a round-robin mapping, i.e., server S_i is i `modulo` *num_servers*. The color of each partition indicates the (color of the) server it resides on. Initially, at time T_0, the directory is small and stored on a single partition $P_0^{(0,1]}$ on server S_0. As the directory grows and the partition size exceeds a threshold number of directory entries, provided this server knows of an underutilized server, S_0 splits $P_0^{(0,1]}$ into two by moving the greater half of its hash-space range to a new partition $P_1^{(0.5,1]}$ on S_1. As the directory expands, servers continue to split partitions onto more servers until all have about the same fraction of the hash-space to manage (analyzed in Section 5.2 and 5.3). GIGA+ computes a split's target partition identifier using well-known radix-based techniques.[3]

The key goal for GIGA+ is for each server to split independently, without system-wide serialization or synchronization. Accordingly, servers make local decisions to split a partition. The side-effect of uncoordinated growth is that GIGA+ servers do not have a global view of the partition-to-server mapping on any one server; each server only has a partial view of the entire index (the mapping tables in Figure 1). Other than the partitions that a server manages, a server knows only the identity of the server that knows more about each "child" partition resulting from a prior split by this server. In Figure 1, at time T_3, server S_1 manages partition P_1 at tree depth $r = 3$, and knows that it previously split P_1 to create children partitions, P_3 and P_5, on servers S_0 and S_2 respectively. Servers are mostly unaware about partition splits that happen on other servers (and did not target them); for instance, at time T_3, server S_0 is unaware of partition P_5 and server S_1 is unaware of partition P_2.

Specifically, each server knows only the split history of its partitions. The full GIGA+ index is a complete history of the directory partitioning, which is the transitive closure over the local mappings on each server. This full index is also not maintained synchronously by any client. GIGA+ clients can enumerate the partitions of a directory by traversing its split histories starting with the zeroth partition P_0. However, such a full index constructed and

[2] For simplicity, we disallow the hash value zero from being used.

[3] GIGA+ calculates the identifier of partition i using the depth of the tree, r, which is derived from the number of splits of the zeroth partition P_0. Specifically, if a partition has an identifier i and is at tree depth r, then in the next split P_i will move half of its filenames, from the larger half of its hash-range, to a new partition with identifier $i+2^r$. After a split completes, both partitions will be at depth $r+1$ in the tree. In Figure 1, for example, partition $P_1^{(0.5,0.75]}$, with identifier $i = 1$, is at tree depth $r = 2$. A split causes P_1 to move the larger half of its hash-space $(0.625, 0.75]$ to the newly created partition P_5, and both partitions are then at tree depth of r = 3.

cached by a client may be stale at any time, particularly for rapidly mutating directories.

3.3 Tolerating inconsistent mapping at clients

Clients seeking a specific filename find the appropriate partition by probing servers, possibly incorrectly, based on their cached index. To construct this index, a client must have resolved the directory's parent directory entry which contains a cluster-wide i-node identifying the server and partition for the zeroth partition P_0. Partition P_0 may be the appropriate partition for the sought filename, or it may not because of a previous partition split that the client has not yet learned about. An "incorrectly" addressed server detects the addressing error by recomputing the partition identifier by re-hashing the filename. If this hashed filename does not belong in the partition it has, this server sends a split history update to the client. The client updates its cached version of the global index and retries the original request.

The drawback of allowing inconsistent indices is that clients may need additional probes before addressing requests to the correct server. The required number of incorrect probes depends on the client request rate and the directory mutation rate (rate of splitting partitions). It is conceivable that a client with an empty index may send $O(log(N_p))$ incorrect probes, where N_p is the number of partitions, but GIGA+'s split history updates makes this many incorrect probes unlikely (described in Section 5.4). Each update sends the split histories of all partitions that reside on a given server, filling all gaps in the client index known to this server and causing client indices to catch up quickly. Moreover, after a directory stops splitting partitions, clients soon after will no longer incur any addressing errors. GIGA+'s eventual consistency for cached indices is different from LH*'s eventual consistency because the latter's idea of independent splitting (called pre-splitting in their paper) suffers addressing errors even when the index stops mutating [36].

3.4 Handling server additions

This section describes how GIGA+ adapts to the addition of servers in a running directory service.[4]

When new servers are added to an existing configuration, the system is immediately no longer load balanced, and it should re-balance itself by migrating a minimal number of directory entries from all existing servers equally. Using the round-robin partition-to-server mapping, shown in Figure 1, a naive server addition scheme would require re-mapping almost all directory entries whenever a new server is added.

GIGA+ avoids re-mapping all directory entries on addition of servers by differentiating the partition-to-server mapping for initial directory growth from the mapping for additional servers. For additional servers, GIGA+ does not use the round-robin partition-to-server map (shown in Figure 1) and instead maps all future partitions to the new servers in a "sequential manner". The benefit of round-robin mapping is faster exploitation of parallelism when a directory is small and growing, while a sequential mapping for the tail set of partitions does not disturb previously mapped partitions more than is mandatory for load balancing.

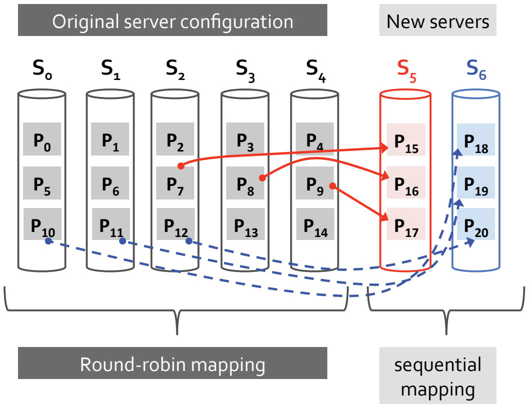


Figure 2 – Server additions in GIGA+. *To minimize the amount of data migrated, indicated by the arrows that show splits, GIGA+ changes the partition-to-server mapping from round-robin on the original server set to sequential on the newly added servers.*

Figure 2 shows an example where the original configuration has 5 servers with 3 partitions each, and partitions P_0 to P_{14} use a round-robin rule (for P_i, server is $i \bmod N$, where N is number of servers). After the addition of two servers, the six new partitions P_{15}-P_{20} will be mapped to servers using the new mapping rule: $i \operatorname{div} M$, where M is the number of partitions per server (e.g., 3 partitions/server).

In GIGA+ even the number of servers can be stale at servers and clients. The arrival of a new server and its order in the global server list is declared by the cluster file system's configuration management protocol, such as Zookeeper for HDFS [29], leading to each existing server eventually noticing the new server. Once it knows about new servers, an existing server can inspect its partitions for those that have sufficient directory entries to warrant splitting and would split to a newly added server. The normal GIGA+ splitting mechanism kicks in to migrate only directory entries that belong on the new servers. The order in which an existing server inspects partitions can be entirely driven by client references to partitions, biasing migration in favor of active directories. Or based on an administrator control, it can also be driven by a background traversal of a list of partitions whose size exceeds the splitting threshold.

[4] Server removal (i.e., decommissioned, not failed and later replaced) is not as important for high performance systems so we leave it to be done by user-level data copy tools.

4 GIGA+ Implementation

GIGA+ indexing mechanism is primarily concerned with distributing the contents and work of large file system directories over many servers, and client interactions with these servers. It is not about the representation of directory entries on disk, and follows the convention of reusing mature local file systems like ext3 or ReiserFS (in Linux) for disk management found as is done by many modern cluster file systems [39, 46, 54, 63, 66].

The most natural implementation strategy for GIGA+ is as an extension of the directory functions of a cluster file system. GIGA+ is not about striping the data of huge files, server failure detection and failover mechanism, or RAID/replication of data for disk fault tolerance. These functions are present and, for GIGA+ purposes, adequate in most cluster file systems. Authors of a new version of PVFS, called OrangeFS, and doing just this by integrating GIGA+ into OrangeFS [2, 45]. Our goal is not to compete with most features of these systems, but to offer technology for advancing their support of high rates of mutation of large collections of small files.

For the purposes of evaluating GIGA+ on file system directory workloads, we have built a skeleton cluster file system; that is, we have not implemented data striping, fault detection or RAID in our experimental framework. Figure 3 shows our user-level GIGA+ directory prototypes built using the FUSE API [19]. Both client and server processes run in user-space, and communicate over TCP using SUN RPC [56]. The prototype has three layers: unmodified applications running on clients, the GIGA+ indexing modules (of the skeletal cluster file system on clients and servers) and a backend persistent store at the server. Applications interact with a GIGA+ client using the VFS API (e.g., `open()`, `creat()` and `close()` syscalls). The FUSE kernel module intercepts and redirects these VFS calls the client-side GIGA+ indexing module which implements the logic described in the previous section.

4.1 Server implementation

The GIGA+ server module's primary purpose is to synchronize and serialize interactions between all clients and a specific partition. It need not "store" the partitions, but it owns them by performing all accesses to them. Our server-side prototype is currently layered on lower level file systems, ext3 and ReiserFS. This decouples GIGA+ indexing mechanisms from on-disk representation.

Servers map logical GIGA+ partitions to directory objects within the backend file system. For a given (huge) directory, its entry in its parent directory names the "zeroth partition", $P_0^{(0,1]}$, which is a directory in a server's underlying file system. Most directories are not huge and will be represented by just this one zeroth partition.

GIGA+ stores some information as extended attributes on the directory holding a partition: a GIGA+ directory ID (unique across servers), the the partition identifier P_i and its range $(x,y]$. The range implies the leaf in the directory's logical tree view of the huge directory associated with this partition (the center column of Figure 1) and that determines the prior splits that had to have occurred to cause this partition to exist (that is, the split history).

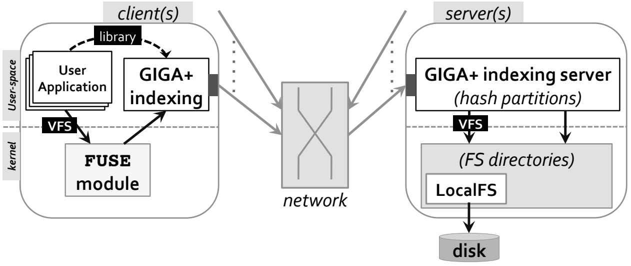


Figure 3 – GIGA+ experimental prototype.

To associate an entry in a cached index (a partition) with a specific server, we need the list of servers over which partitions are round robin allocated and the list of servers over which partitions are sequentially allocated. The set of servers that are known to the cluster file system at the time of splitting the zeroth partition is the set of servers that are round robin allocated for this directory and the set of servers that are added after a zeroth partition is split are the set of servers that are sequentially allocated.[5]

Because the current list of servers will always be available in a cluster file system, only the list of servers at the time of splitting the zeroth server needs to be also stored in a partition's extended attributes. Each split propagates the directory ID and set of servers at the time of the zeroth partition split to the new partition, and sets the new partition's identifier P_i and range $(x,y]$ as well as providing the entries from the parent partition that hash into this range $(x,y]$.

Each partition split is handled by the GIGA+ server by locally locking the particular directory partition, scanning its entries to build two sub-partitions, and then transactionally migrating ownership of one partition to another server before releasing the local lock on the partition [55]. In our prototype layered on local file systems, there is no transactional migration service available, so we move the directory entries and copy file data between servers. Our experimental splits are therefore more expensive than they should be in a production cluster file system.

4.2 Client implementation

The GIGA+ client maintains cached information, some potentially stale, global to all directories. It caches the current server list (which we assume only grows over time)

[5] The contents of a server list are logical server IDs (or names) that are converted to IP addresses dynamically by a directory service integrated with the cluster file system. Server failover (and replacement) will bind a different address to the same server ID so the list does not change during normal failure handling.

and the number of partitions per server (which is fixed) obtained from whichever server GIGA+ was mounted on. For each active directory GIGA+ clients cache the cluster-wide i-node of the zeroth partition, the directory ID, and the number of servers at the time when the zeroth partition first split. The latter two are available as extended attributes of the zeroth partition. Most importantly, the client maintains a bitmap of the global index built according to Section 3, and a maximum tree-depth, $r = \lceil log(i) \rceil$, of any partition P_i present in the global index.

Searching for a file name with a specific hash value, H, is done by inspecting the index bitmap at the offset j determined by the r lower-order bits of H. If this is set to '1' then H is in partition P_j. If not, decrease r by one and repeat until $r = 0$ which refers to the always known zeroth partition P_0. Identifying the server for partition P_j is done by lookup in the current server list. It is either $j\texttt{mod}N$, where N is the number of servers at the time the zeroth partition split), or $j\texttt{div}M$, where M is the number of partitions per server, with the latter used if j exceeds the product of the number of servers at the time of zeroth partition split and the number of partitions per server.

Most VFS operations depend on lookups; `readdir()` however can be done by walking the bitmaps, enumerating the partitions and scanning the directories in the underlying file system used to store partitions.

4.3 Handling failures

Modern cluster file systems scale to sizes that make fault tolerance mandatory and sophisticated [8, 20, 65]. With GIGA+ integrated in a cluster file system, fault tolerance for data and services is already present, and GIGA+ does not add major challenges. In fact, handling network partitions and client-side reboots are relatively easy to handle because GIGA+ tolerates stale entries in a client's cached index of the directory partition-to-server mapping and because GIGA+ does not cache directory entries in client or server processes (changes are written through to the underlying file system). Directory-specific client state can be reconstructed by contacting the zeroth partition named in a parent directory entry, re-fetching the current server list and rebuilding bitmaps through incorrect addressing of server partitions during normal operations.

Other issues, such as on-disk representation and disk failure tolerance, are a property of the existing cluster file system's directory service, which is likely to be based on replication even when large data files are RAID encoded [66]. Moreover, if partition splits are done under a lock over the entire partition, which is how our experiments are done, the implementation can use a non-overwrite strategy with a simple atomic update of which copy is live. As a result, recovery becomes garbage collection of spurious copies triggered by the failover service when it launches a new server process or promotes a passive backup to be the active server [9, 29, 65].

While our architecture presumes GIGA+ is integrated into a full featured cluster file system, it is possible to layer GIGA+ as an interposition layer over and independent of a cluster file system, which itself is usually layered over multiple independent local file systems [20, 46, 54, 66]. Such a layered GIGA+ would not be able to reuse the fault tolerance services of the underlying cluster file system, leading to an extra layer of fault tolerance. The primary function of this additional layer of fault tolerance is replication of the GIGA+ server's write-ahead logging for changes it is making in the underlying cluster file system, detection of server failure, election and promotion of backup server processes to be primaries, and reprocessing of the replicated write-ahead log. Even the replication of the write-ahead log may be unnecessary if the log is stored in the underlying cluster file system, although such logs are often stored outside of cluster file systems to improve the atomicity properties writing to them [12, 26]. To ensure load balancing during server failure recovery, the layered GIGA+ server processes could employ the well-known chained-declustering replication mechanism to shift work among server processes [28], which has been used in other distributed storage systems [33, 60].

5 Experimental Evaluation

Our experimental evaluation answers two questions: (1) How does GIGA+ scale? and (2) What are the tradeoffs of GIGA+'s design choices involving incremental growth, weak index consistency and selection of the underlying local file system for out-of-core indexing (when partitions are very large)?

All experiments were performed on a cluster of 64 machines, each with dual quad-core 2.83GHz Intel Xeon processors, 16GB memory and a 10GigE NIC, and Arista 10 GigE switches. All nodes were running the Linux 2.6.32-js6 kernel (Ubuntu release) and GIGA+ stores partitions as directories in a local file system on one 7200rpm SATA disk (a different disk is used for all non-GIGA+ storage). We assigned 32 nodes as servers and the remaining 32 nodes as load generating clients. The threshold for splitting a partition is always 8,000 entries.

We used the synthetic `mdtest` benchmark [41] (used by parallel file system vendors and users) to insert zero-byte files in to a directory [27, 63]. We generated three types of workloads. First, a *concurrent create* workload that creates a large number of files concurrently in a single directory. Our configuration uses eight processes per client to simultaneously create files in a common directory, and the number of files created is proportional to the number of servers: a single server manages 400,000 files, a 800,000 file directory is created on 2 servers, a 1.6 million file directory on 4 servers, up to a 12.8 million file directory on 32 servers. Second, we use a *lookup workload* that performs a `stat()` on random files in the directory. And

	File System	File creates/second in one directory
GIGA+	Library API	17,902
(layered on Reiser)	VFS/FUSE API	5,977
Local	Linux ext3	16,470
file systems	Linux ReiserFS	20,705
Networked	NFSv3 filer	521
file systems	HadoopFS	4,290
	PVFS	1,064

Table 1 – File create rate in a single directory on a single server. *An average of five runs (with 1% standard deviation).*

finally, we use a mixed workload where clients issue create and lookup requests in a pre-configured ratio.

5.1 Scale and performance

We begin with a baseline for the performance of various file systems running the `mdtest` benchmark. First we compare `mdtest` running locally on Linux ext3 and ReiserFS local file systems to `mdtest` running on a separate client and single server instance of the PVFS cluster file system (using ext3) [46], Hadoop's HDFS (using ext3) [54] and a mature commercial NFSv3 filer. In this experiment GIGA+ always uses one partition per server. Table 1 shows the baseline performance.

For GIGA+ we use two machines with ReiserFS on the server and two ways to bind `mdtest` to GIGA+: direct library linking (non-POSIX) and VFS/FUSE linkage (POSIX). The library approach allows `mdtest` to use custom object creation calls (such as `giga_creat()`) avoiding system call and FUSE overhead in order to compare to `mdtest` directly in the local file system. Among the local file systems, with local `mdtest` threads generating file creates, both ReiserFS and Linux ext3 deliver high directory insert rates.[6] Both file systems were configured with `-noatime` and `-nodiratime` option; Linux ext3 used write-back journaling and the `dir_index` option to enable hashed-tree indexing, and ReiserFS was configured with the `-notail` option, a small-file optimization that packs the data inside an i-node for high performance [47]. GIGA+ with `mdtest` workload generating threads on a different machine, when using the library interface (sending only one RPC per create) and ReiserFS as the backend file system, creates at better than 80% of the rate of ReiserFS with local load generating threads. This comparison shows that remote RPC is not a huge penalty for GIGA+. We tested this library version only to gauge GIGA+ efficiency compared to local file systems and do not use this setup for any remaining experiments.

To compare with the network file systems, GIGA+ uses the VFS/POSIX interface. In this case each VFS

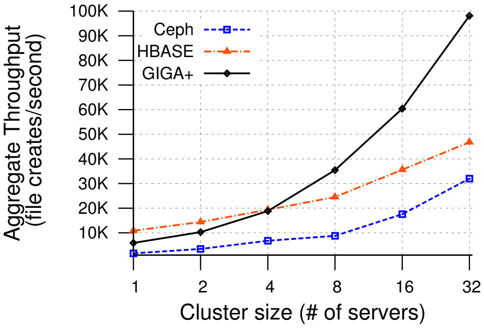


Figure 4 – Scalability of GIGA+ FS directories. GIGA+ *directories deliver a peak throughput of roughly 98,000 file creates per second. The behavior of underlying local file system (ReiserFS) limits* GIGA+*'s ability to match the ideal linear scalability.*

file `creat()` results in three RPC calls to the server: `getattr()` to check if a file exists, the actual `creat()` and another `getattr()` after creation to load the created file's attributes. For a more enlightening comparison, cluster file systems were configured to be functionally equivalent to the GIGA+ prototype; specifically, we disabled HDFS's write-ahead log and replication, and we used default PVFS which has no redundancy unless a RAID controller is added. For the NFSv3 filer, because it was in production use, we could not disable its RAID redundancy and it is correspondingly slower than it might otherwise be. GIGA+ directories using the VFS/FUSE interface also outperforms all three networked file systems, probably because the GIGA+ experimental prototype is skeletal while others are complex production systems.

Figure 4 plots aggregate operation throughput, in file creates per second, averaged over the complete *concurrent create* benchmark run as a function of the number of servers (on a log-scale X-axis). GIGA+ with partitions stored as directories in ReiserFS scales linearly up to the size of our 32-server configuration, and can sustain 98,000 file creates per second - this exceeds today's most rigorous scalability demands [44].

Figure 4 also compares GIGA+ with the scalability of the Ceph file system and the HBase distributed key-value store. For Ceph, Figure 4 reuses numbers from experiments performed on a different cluster from the original paper [63]. That cluster used dual-core 2.4GHz machines with IDE drives, with equal numbered separate nodes as workload generating clients, metadata servers and disk servers with object stores layered on Linux ext3. HBase is used to emulate Google's Colossus file system which plans to store file system metadata in BigTable instead of internally on single master node[18]. We setup HBase on a 32-node HDFS configuration with a single copy (no replication) and disabled two parameters: blocking while the HBase servers are doing compactions and write-ahead logging for inserts (a common practice to speed up insert-

[6] We tried XFS too, but it was extremely slow during the create-intensive workload and do not report those numbers in this paper.

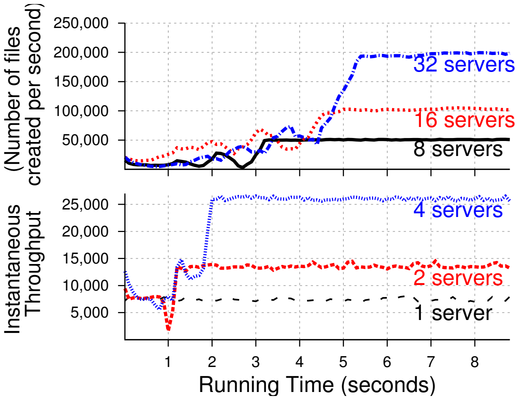


Figure 5 – Incremental scale-out growth. GIGA+ *achieves linear scalability after distributing one partition on each available server. During scale-out, periodic drops in aggregate create rate correspond to concurrent splitting on all servers.*

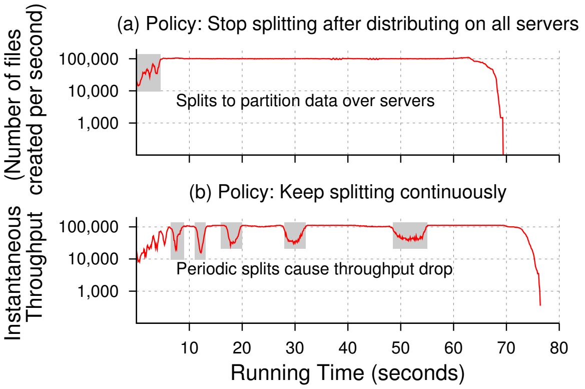


Figure 6 – Effect of splitting heuristics. GIGA+ *shows that splitting to create at most one partition on each of the 16 servers delivers scalable performance. Continuous splitting, as in classic database indices, is detrimental in a distributed scenario.*

ing data in HBase). This configuration allowed HBase to deliver better performance than GIGA+ for the single server configuration because the HBase tables are striped over all 32-nodes in the HDFS cluster. But configurations with many HBase servers scale poorly.

GIGA+ also demonstrated scalable performance for the *concurrent lookup* workload delivering a throughput of more than 600,000 file lookups per second for our 32-server configuration (not shown). Good lookup performance is expected because the index is not mutating and load is well-distributed among all servers; the first few lookups fetch the directory partitions from disk into the buffer cache and the disk is not used after that. Section 5.4 gives insight on addressing errors during mutations.

5.2 Incremental scaling properties

In this section, we analyze the scaling behavior of the GIGA+ index independent of the disk and the on-disk directory layout (explored later in Section 5.5). To eliminate performance issues in the disk subsystem, we use Linux's in-memory file system, `tmpfs`, to store directory partitions. Note that we use `tmpfs` only in this section, all other analysis uses on-disk file systems.

We run the *concurrent create* benchmark to create a large number of files in an empty directory and measure the aggregate throughput (file creates per second) continuously throughout the benchmark. We ask two questions about scale-out heuristics: (1) what is the effect of splitting during incremental scale-out growth? and (2) how many partitions per server do we keep?

Figure 5 shows the first 8 seconds of the *concurrent create* workload when the number of partitions per server is one. The primary result in this figure is the near linear create rate seen after the initial seconds. But the initial few seconds are more complex. In the single server case, as expected, the throughput remains flat at roughly 7,500 file creates per second due to the absence of any other server. In the 2-server case, the directory starts on a single server and splits when it has more than 8,000 entries in the partition. When the servers are busy splitting, at the 0.8-second mark, throughput drops to half for a short time.

Throughput degrades even more during the scale-out phase as the number of directory servers goes up. For instance, in the 8-server case, the aggregate throughput drops from roughly 25,000 file creates/second at the 3-second mark to as low as couple of hundred creates/second before growing to the desired 50,000 creates/second. This happens because all servers are busy splitting, i.e., partitions overflow at about the same time which causes all servers (where these partitions reside) to split without any co-ordination at the same time. And after the split spreads the directory partitions on twice the number of servers, the aggregate throughput achieves the desired linear scale.

In the context of the second question about how many partitions per server, classic hash indices, such as extendible and linear hashing [17, 34], were developed for out-of-core indexing in single-node databases. An out-of-core table keeps splitting partitions whenever they overflow because the partitions correspond to disk allocation blocks [23]. This implies an unbounded number of partitions per server as the table grows. However, the splits in GIGA+ are designed to parallelize access to a directory by distributing the directory load over all servers. Thus GIGA+ can stop splitting after each server has a share of work, i.e., at least one partition. When GIGA+ limits the number of partitions per server, the size of partitions continue to grow and GIGA+ lets the local file system on each server handle physical allocation and out-of-core memory management.

Figure 6 compares the effect of different policies for the number of partitions per server on the system throughput (using a log-scale Y-axis) during a test in which a large directory is created over 16 servers. Graph (a) shows a split policy that stops when every server has one partition, caus-

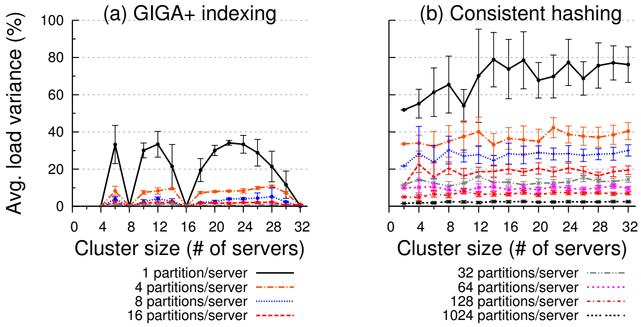


Figure 7 – Load-balancing in GIGA+. *These graphs show the quality of load balancing measured as the mean load deviation across the entire cluster (with 95% confident interval bars). Like virtual servers in consistent hashing,* GIGA+ *also benefits from using multiple hash partitions per server.* GIGA+ *needs one to two orders of magnitude fewer partitions per server to achieve comparable load distribution relative to consistent hashing.*

ing partitions to ultimately get much bigger than 8,000 entries. Graph (b) shows the continuous splitting policy used by classic database indices where a split happens whenever a partition has more than 8,000 directory entries. With continuous splitting the system experiences periodic throughput drops that last longer as the number of partitions increases. This happens because repeated splitting maps multiple partitions to each server, and since uniform hashing will tend to overflow all partitions at about the same time, multiple partitions will split on all the servers at about the same time.

Lesson #1: To avoid the overhead of continuous splitting in a distributed scenario, GIGA+ stops splitting a directory after all servers have a fixed number of partitions and lets a server's local file system deal with out-of-core management of large partitions.

5.3 Load balancing efficiency

The previous section showed only configurations where the number of servers is a power-of-two. This is a special case because it is naturally load-balanced with only a single partition per server: the partition on each server is responsible for a hash-range of size 2^r-th part of the total hash-range $(0,1]$. When the number of servers is not a power-of-two, however, there is load imbalance. Figure 7 shows the load imbalance measured as the average fractional deviation from even load for all numbers of servers from 1 to 32 using Monte Carlo model of load distribution. In a cluster of 10 servers, for example, each server is expected to handle 10% of the total load; however, if two servers are experiencing 16% and 6% of the load, then they have 60% and 40% variance from the average load respectively. For different cluster sizes, we measure the variance of each server, and use the average (and 95% confidence interval error bars) over all the servers.

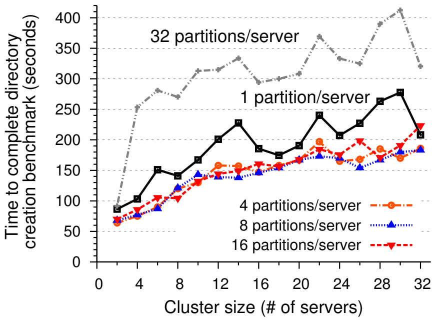


Figure 8 – Cost of splitting partitions. *Using 4, 8, or 16 partitions per server improves the performance of* GIGA+ *directories layered on Linux ext3 relative to 1 partition per server (better load-balancing) or 32 partitions per server (when the cost of more splitting dominates the benefit of load-balancing).*

We compute load imbalance for GIGA+ in Figure 7(a) as follows: when the number of servers N is not a power-of-two, $2^r < N < 2^{r+1}$, then a random set of $N-2^r$ partitions from tree depth r, each with range size $1/2^r$, will have split into $2(N-2^r)$ partitions with range size $1/2^{r+1}$. Figure 7(a) shows the results of five random selections of $N-2^r$ partitions that split on to the $r+1$ level. Figure 7(a) shows the expected periodic pattern where the system is perfectly load-balanced when the number of servers is a power-of-two. With more than one partition per server, each partition will manage a smaller portion of the hash-range and the sum of these smaller partitions will be less variable than a single large partition as shown in Figure 7(a). Therefore, more splitting to create more than one partition per server significantly improves load balance when the number of servers is not a power-of-two.

Multiple partitions per server is also used by Amazon's Dynamo key-value store to alleviate the load imbalance in consistent hashing [15]. Consistent hashing associates each partition with a random point in the hash-space $(0,1]$ and assigns it the range from this point up to the next larger point and wrapping around, if necessary. Figure 7(b) shows the load imbalance from Monte Carlo simulation of using multiple partitions (virtual servers) in consistent hashing by using five samples of a random assignment for each partition and how the sum, for each server, of partition ranges selected this way varies across servers. Because consistent hashing's partitions have more randomness in each partition's hash-range, it has a higher load variance than GIGA+ – almost two times worse. Increasing the number of hash-range partitions significantly improves load distribution, but consistent hashing needs more than 128 partitions per server to match the load variance that GIGA+ achieves with 8 partitions per server – an order of magnitude more partitions.

More partitions is particularly bad because it takes longer for the system to stop splitting, and Figure 8 shows how this can impact overall performance. Consistent hash-

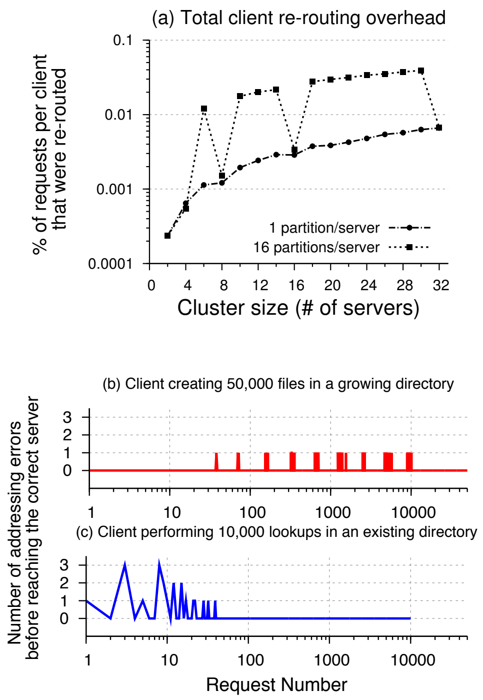


Figure 9 – Cost of using inconsistent mapping at the clients. *Using weak consistency for mapping state at the clients has a very negligible overhead on client performance (a). And this overhead – extra message re-addressing hops – occurs for initial requests until the client learns about all the servers (b and c).*

ing theory has alternate strategies for reducing imbalance but these often rely on extra accesses to servers all of the time and global system state, both of which will cause impractical degradation in our system [10].

Since having more partitions per server always improves load-balancing, at least a little, how many partitions should GIGA+ use? Figure 8 shows the *concurrent create* benchmark time for GIGA+ as a function of the number of servers for 1, 4, 8, 16 and 32 partitions per server. We observe that with 32 partitions per server GIGA+ is roughly 50% slower than with 4, 8 and 16 partitions per server. Recall from Figure 7(a) that the load-balancing efficiency from using 32 partitions per server is only about 1% better than using 16 partitions per server; the high cost of splitting to create twice as many partitions outweighs the minor load-balancing improvement.

Lesson #2: Splitting to create more than one partition per server significantly improves GIGA+ load balancing for non power-of-two numbers of servers, but because of the performance penalty during extra splitting the overall performance is best with only a few partitions per server.

5.4 Cost of weak mapping consistency

Figure 9(a) shows the overhead incurred by clients when their cached indices become stale. We measure the percentage of all client requests that were re-routed when running the *concurrent create* benchmark on different cluster sizes. This figure shows that, in absolute terms, fewer than 0.05% of the requests are addressed incorrectly; this is only about 200 requests per client because each client is doing 400,000 file creates. The number of addressing errors increases proportionally with the number of partitions per server because it takes longer to create all partitions. In the case when the number of servers is a power-of-two, after each server has at least one partition, subsequent splits yield two smaller partitions on the same server, which will not lead to any additional addressing errors.

We study further the worst case in Figure 9(a), 30 servers with 16 partitions per server, to learn when addressing errors occur. Figure 9(b) shows the number of errors encountered by each request generated by one client thread (i.e., one of the eight workload generating threads per client) as it creates 50,000 files in this benchmark. Figure 9(b) suggests three observations. First, the index update that this thread gets from an incorrectly addressed server is always sufficient to find the correct server on the second probe. Second, that addressing errors are bursty, one burst for each level of the index tree needed to create 16 partitions on each of 30 servers, or 480 partitions ($2^8 < 480 < 2^9$). And finally, that the last 80% of the work is done after the last burst of splitting without any addressing errors.

To further emphasize how little incorrect server addressing clients generate, Figure 9(c) shows the addressing experience of a new client issuing 10,000 lookups after the current create benchmark has completed on 30 servers with 16 partitions per server.[7] This client makes no more than 3 addressing errors for a specific request, and no more than 30 addressing errors total and makes no more addressing errors after the 40th request.

Lesson #3: GIGA+ clients incur neglible overhead (in terms of incorrect addressing errors) due to stale cached indices, and no overhead shortly after the servers stop splitting partitions. Although not a large effect, fewer partitions per server lowers client addressing errors.

5.5 Interaction with backend file systems

Because some cluster file systems represent directories with equivalent directories in a local file system [39] and because our GIGA+ experimental prototype represents partitions as directories in a local file system, we study how the design and implementation of Linux ext3 and ReiserFS local file systems affects GIGA+ partition splits. Although different local file system implementations can be expected to have different performance, especially for emerging workloads like ours, we were surprised by the size of the differences.

Figure 10 shows GIGA+ file create rates when there are 16 servers for four different configurations: Linux ext3

[7] Figure 9 predicts the addressing errors of a client doing only lookups on a mutating directory because both `create(filename)` and `lookup(filename)` do the same addressing.

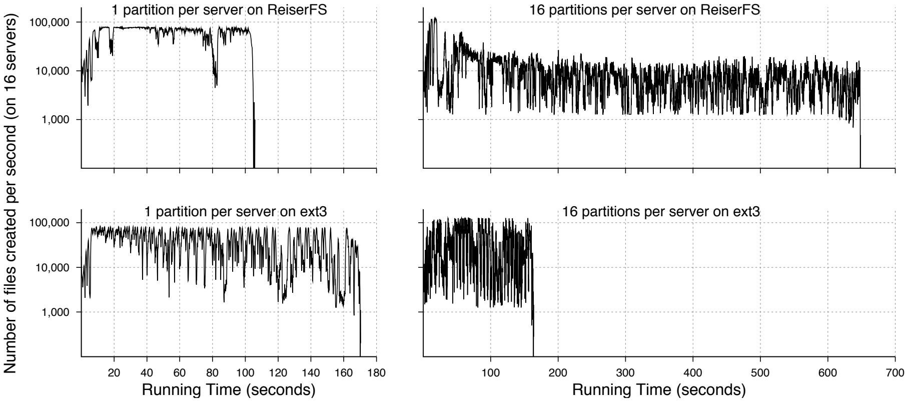


Figure 10 – Effect of underlying file systems. *This graph shows the concurrent create benchmark behavior when the* GIGA+ *directory service is distributed on 16 servers with two local file systems, Linux ext3 and ReiserFS. For each file system, we show two different numbers of partitions per server, 1 and 16.*

or ReiserFS storing partitions as directories, and 1 or 16 partitions per server. Linux ext3 directories use h-trees [11] and ReiserFS uses balanced B-trees [47]. We observed two interesting phenomenon: first, the benchmark running time varies from about 100 seconds to over 600 seconds, a factor of 6, and second, the backend file system yielding the faster performance is different when there are 16 partitions on each server than with only one.

Comparing a single partition per server in GIGA+ over ReiserFS and over ext3 (left column in Figure 10), we observe that the benchmark completion time increases from about 100 seconds using ReiserFS to nearly 170 seconds using ext3. For comparison, the same benchmark completed in 70 seconds when the backend was the in-memory `tmpfs` file system. Looking more closely at Linux ext3, as a directory grows, ext3's journal also grows and periodically triggers ext3's `kjournald` daemon to flush a part of the journal to disk. Because directories are growing on all servers at roughly the same rate, multiple servers flush their journal to disk at about the same time leading to troughs in the aggregate file create rate. We observe this behavior for all three journaling modes supported by ext3. We confirmed this hypothesis by creating an ext3 configuration with the journal mounted on a second disk in each server, and this eliminated most of the throughput variability observed in ext3, completing the benchmark almost as fast as with ReiserFS. For ReiserFS, however, placing the journal on a different disk had little impact.

The second phenomenon we observe, in the right column of Figure 10, is that for GIGA+ with 16 partitions per server, ext3 (which is insensitive to the number of partitions per server) completes the create benchmark more than four times faster than ReiserFS. We suspect that this results from the on-disk directory representation. ReiserFS uses a balanced B-tree for *all objects* in the file system, which re-balances as the file system grows and changes over time [47]. When partitions are split more often, as in case of 16 partitions per server, the backend file system structure changes more, which triggers more re-balancing in ReiserFS and slows the create rate.

Lesson #4: Design decisions of the backend file system have subtle but large side-effects on the performance of a distributed directory service. Perhaps the representation of a partition should not be left to the vagaries of whatever local file system is available.

6 Conclusion

In this paper we address the emerging requirement for POSIX file system directories that store massive number of files and sustain hundreds of thousands of concurrent mutations per second. The central principle of GIGA+ is to use asynchrony and eventual consistency in the distributed directory's internal metadata to push the limits of scalability and concurrency of file system directories. We used these principles to prototype a distributed directory implementation that scales linearly to best-in-class performance on a 32-node configuration. Our analysis also shows that GIGA+ achieves better load balancing than consistent hashing and incurs a neglible overhead from allowing stale lookup state at its clients.

Acknowledgements. This work is based on research supported in part by the Department of Energy, under award number DE-FC02-06ER25767, by the Los Alamos National Laboratory, under contract number 54515-001-07, by the Betty and Gordon Moore Foundation, by the National Science Foundation under awards CCF-1019104 and SCI-0430781, and by Google and Yahoo! research awards. We thank Cristiana Amza, our shepherd, and the anonymous reviewers for their thoughtful reviews of our paper. John Bent and Gary Grider from LANL provided valuable feedback from the early stages of this work; Han Liu assisted with HBase experimental setup; and Vijay Vasudevan, Wolfgang Richter, Jiri Simsa, Julio Lopez and Varun Gupta helped with early drafts of the paper. We also thank the member companies of the PDL Consortium (including APC, DataDomain, EMC, Facebook, Google, Hewlett-Packard, Hitachi, IBM, Intel, LSI, Microsoft, NEC, NetApp, Oracle, Seagate, Sun, Symantec, and VMware) for their interest, insights, feedback, and support.

References

[1] Private Communication with Frank Schmuck and Roger Haskin, IBM, February 2010.

[2] Private Communication with Walt Ligon, OrangeFS (`http://orangefs.net`), November 2010.

[3] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Silberschatz, and A. Rasin. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. In *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB '09)*, Lyon, France, August 2009.

[4] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A Five-Year Study of File-System Metadata. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST '07)*, San Jose CA, February 2007.

[5] R. Agrawal, A. Ailamaki, P. A. Bernstein, E. A. Brewer, M. J. Carey, S. Chaudhuri, A. Doan, D. Florescu, M. J. Franklin, H. Garcia-Molina, J. Gehrke, L. Gruenwald, L. M. Haas, A. Y. Halevy, J. M. Hellerstein, Y. E. Ioannidis, H. F. Korth, D. Kossmann, S. Madden, R. Magoulas, B. C. Ooi, T. O'Reilly, R. Ramakrishnan, S. Sarawagi, M. Stonebraker, A. S. Szalay, and G. Weikum. The Claremont report on database research. *ACM SIGMOD Record*, 37(3), September 2008.

[6] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a Needle in Haystack: Facebook's Photo Storage. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, Canada, October 2010.

[7] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. PLFS: A Checkpoint Filesystem for Parallel Applications. In *Proceedings of the ACM/IEEE Transactions on Computing Conference on High Performance Networking and Computing (SC '09)*, Portland OR, November 2009.

[8] P. Braam and B. Neitzel. Scalable Locking and Recovery for Network File Systems. In *Proceedings of the 2nd International Petascale Data Storage Workshop (PDSW '07)*, Reno NV, November 2007.

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

[10] J. Byers, J. Considine, and M. Mitzenmacher. Simple Load Balancing for Distributed Hash Tables. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Berkeley CA, February 2003.

[11] M. Cao, T. Y. Ts'o, B. Pulavarty, S. Bhattacharya, A. Dilger, and A. Tomas. State of the Art: Where we are with the ext3 filesystem. In *Proceedings of the Ottawa Linux Symposium (OLS '07)*, Ottawa, Canada, June 2007.

[12] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle WA, November 2006.

[13] 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)*, Banff, Canada, October 2001.

[14] S. Dayal. Characterizing HEC Storage Systems at Rest. Technical Report CMU-PDL-08-109, Carnegie Mellon University, July 2008.

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, Stevenson WA, October 2007.

[16] J. R. Douceur and J. Howell. Distributed Directory Service in the Farsite File System. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle WA, November 2006.

[17] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible Hashing – A Fast Access Method for Dynamic Files. *ACM Transactions on Database Systems*, 4(3), September 1979.

[18] A. Fikes. Storage Architecture and Challenges. Presentation at the 2010 Google Faculty Summit. Talk slides at `http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en/us/university/relations/facultysummit2010/storage_architecture_and_challenges.pdf`.

[19] FUSE. Filesystem in Userspace. `http://fuse.sf.net/`.

[20] S. Ghemawat, H. Gobioff, and S.-T. Lueng. Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing NY, October 2003.

[21] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A Cost-Effective, High-Bandwidth Storage Architecture. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '98)*, San Jose CA, October 1998.

[22] GPFS. An Introduction to GPFS Version 3.2.1. `http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp`, November 2008.

[23] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1992.

[24] S. Gribble, E. Brewer, J. Hellerstein, and D. Culler. Scalable Distributed Data Structures for Internet Service Construction. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI '00)*, San Diego CA, October 2000.

[25] J. H. Hartman and J. K. Ousterhout. The Zebra Striped Network File System. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, Asheville NC, December 1993.

[26] HBase. The Hadoop Database. `http://hadoop.apache.org/hbase/`, December 2010.

[27] R. Hedges, K. Fitzgerald, M. Gary, and D. M. Stearman. Comparison of leading parallel NAS file systems on commodity hardware. Poster at the Petascale Data Storage Workshop 2010. `http://www.pdsi-scidac.org/events/PDSW10/resources/posters/parallelNASFSs.pdf`, November 2010.

[28] H.-I. Hsaio and D. J. DeWitt. Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines. In *Proceedings of the 6th International Conference on Data Engineering (ICDE '90)*, Washington D.C., February 1990.

[29] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '10)*, Boston MA, June 2010.

[30] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the ACM Symposium on Theory of Computing (STOC '97)*, El Paso TX, May 1997.

[31] P. Kogge. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. DARPA IPTO Report at `http://www.er.doe.gov/ascr/Research/CS/DARPAexascale-hardware(2008).pdf`, September 2008.

[32] A. Lakshman and P. Malik. Cassandra - A Decentralized Structured Storage System. In *Proceedings of the Workshop on Large-Scale Distribued Systems and Middleware (LADIS '09)*, Big Sky MT, October 2009.

[33] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks.

In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '96)*, Cambridge MA, October 1996.

[34] W. Litwin. Linear Hashing: A New Tool for File and Table Addressing. In *Proceedings of the 6th International Conference on Very Large Data Bases (VLDB '80)*, Montreal, Canada, October 1980.

[35] W. Litwin, M.-A. Neimat, and D. A. Schneider. LH* - Linear Hashing for Distributed Files. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD '93)*, Washington D.C., June 1993.

[36] W. Litwin, M.-A. Neimat, and D. A. Schneider. LH* - A Scalable, Distributed Data Structure. *ACM Transactions on Database Systems*, 21(4), December 1996.

[37] Lustre. Clustered Metadata Design. `http://wiki.lustre.org/images/d/db/HPCS_CMD_06_15_09.pdf`, September 2009.

[38] Lustre. Clustered Metadata. `http://wiki.lustre.org/index.php/Clustered_Metadata`, September 2010.

[39] Lustre. Lustre File System. `http://www.lustre.org`, December 2010.

[40] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the Foundation for Storage Infrastructure. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco CA, November 2004.

[41] MDTEST. mdtest: HPC benchmark for metadata performance. `http://sourceforge.net/projects/mdtest/`, December 2010.

[42] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A Read/Write Peer-to-peer File System. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston MA, November 2002.

[43] NetApp-Community-Form. Millions of files in a single directory. Discussion at `http://communities.netapp.com/thread/7190?tstart=0`, February 2010.

[44] H. Newman. HPCS Mission Partner File I/O Scenarios, Revision 3. `http://wiki.lustre.org/images/5/5a/Newman_May_Lustre_Workshop.pdf`, November 2008.

[45] OrangeFS. Distributed Directories in OrangeFS v2.8.3-EXP. `http://orangefs.net/trac/orangefs/wiki/Distributeddirectories`.

[46] PVFS2. Parallel Virtual File System, Version 2. `http://www.pvfs2.org`, December 2010.

[47] H. Reiser. ReiserFS. `http://www.namesys.com/`.

[48] 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.

[49] R. A. Rivest. The MD5 Message Digest Algorithm. Internet RFC 1321, April 1992.

[50] R. Ross, E. Felix, B. Loewe, L. Ward, J. Nunez, J. Bent, E. Salmon, and G. Grider. High End Computing Revitalization Task Force (HECRTF), Inter Agency Working Group (HECIWG) File Systems and I/O Research Guidance Workshop 2006. `http://institutes.lanl.gov/hec-fsio/docs/HECIWG-FSIO-FY06-Workshop-Document-FINAL6.pdf`, 2006.

[51] 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, Canada, October 2001.

[52] 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.

[53] M. Seltzer. Beyond Relational Databases. *Communications of the ACM*, 51(7), July 2008.

[54] K. Shvachko, H. Huang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proceedings of the 26th IEEE Transactions on Computing Symposium on Mass Storage Systems and Technologies (MSST '10)*, Lake Tahoe NV, May 2010.

[55] S. Sinnamohideen, R. R. Sambasivan, J. Hendricks, L. Liu, and G. R. Ganger. A Transparently-Scalable Metadata Service for the Ursa Minor Storage System. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '10)*, Boston MA, June 2010.

[56] R. Srinivasan. RPC: Remote Procedure Call Protocol Specification Version 2. Internet RFC 1831, August 1995.

[57] StackOverflow. Millions of small graphics files and how to overcome slow file system access on XP. Discussion at `http://stackoverflow.com/questions/1638219/`, October 2009.

[58] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '01)*, San Diego CA, August 2001.

[59] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. R. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-Store: A Column-Oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB '05)*, Trondheim, Norway, September 2005.

[60] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A Scalable Distributed File System. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, Saint-Malo, France, October 1997.

[61] TOP500. Top 500 Supercomputer Sites. `http://www.top500.org`, December 2010.

[62] D. Tweed. One usage of up to a million files/directory. Email thread at `http://leaf.dragonflybsd.org/mailarchive/kernel/2008-11/msg00070.html`, November 2008.

[63] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle WA, November 2006.

[64] S. A. Weil, K. Pollack, S. A. Brandt, and E. L. Miller. Dynamic Metadata Management for Petabyte-Scale File Systems. In *Proceedings of the ACM/IEEE Transactions on Computing Conference on High Performance Networking and Computing (SC '04)*, Pittsburgh PA, November 2004.

[65] B. Welch. Integrated System Models for Reliable Petascale Storage Systems. In *Proceedings of the 2nd International Petascale Data Storage Workshop (PDSW '07)*, Reno NV, November 2007.

[66] 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 *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST '08)*, San Jose CA, February 2008.

[67] R. Wheeler. One Billion Files: Scalability Limits in Linux File Systems. Presentation at LinuxCon '10. Talk Slides at `http://events.linuxfoundation.org/slides/2010/linuxcon2010_wheeler.pdf`, August 2010.

[68] ZFS-discuss. Million files in a single directory. Email thread at `http://mail.opensolaris.org/pipermail/zfs-discuss/2009-October/032540.html`, October 2009.

AONT-RS: Blending Security and Performance in Dispersed Storage Systems

Jason K. Resch
Development
Cleversafe, Inc.
222 S. Riverside Plaza, Suite 1700
Chicago, IL 60606
jresch@cleversafe.com

James S. Plank
EECS Department
University of Tennessee
203 Claxton Complex
Knoxville, TN 37996
plank@cs.utk.edu

Abstract

Dispersing files across multiple sites yields a variety of obvious benefits, such as availability, proximity and reliability. Less obviously, it enables security to be achieved without relying on encryption keys. Standard approaches to dispersal either achieve very high security with correspondingly high computational and storage costs, or low security with lower costs. In this paper, we describe a new dispersal scheme, called *AONT-RS*, which blends an All-Or-Nothing Transform with Reed-Solomon coding to achieve high security with low computational and storage costs. We evaluate this scheme both theoretically and as implemented with standard open source tools. AONT-RS forms the backbone of a commercial dispersed storage system, which we briefly describe and then use as a further experimental testbed. We conclude with details of actual deployments.

1 Introduction

Dispersed storage systems coalesce multiple storage sites into a collective whole. Files are decomposed into smaller blocks which are computationally massaged and then dispersed to the storage sites. When a client desires to read a file, it retrieves some subset of the blocks, which are combined to reconstitute the original file. Compared to traditional single-site storage systems, dispersed storage systems offer a variety of benefits. Multiple independent storage sites offer greater availability than a single site, since they have no single point of failure. When sites are physically distributed across a wide area, they offer physical proximity to distributed clients, which can improve performance and scalability. Finally, the massaging of data typically includes adding redundancy in the form of erasure codes or secret sharing, which improves reliability in the face of failures.

There have been many dispersed storage systems developed in the past ten years. Examples include storage systems such as Oceanstore [23], Pergamum [29], POTSHARDS [30], PASIS [9], Gridsharing [31], Glacier [11], Cleversafe [4] and Tahoe-LAFS [32] among others. Related to dispersed storage systems are distributed or peer-to-peer storage systems which use replication rather than coding to achieve reliability. Examples include LOCKSS [14], Google file system [8], Elephant [27], PAST [26] and BitTorrent [5].

A side benefit of dispersal is the ability to provide security without the use of encryption keys. The basic techniques are classics from computer science literature: Shamir's secret sharing [28] and Rabin's information dispersal based on non-systematic erasure codes [21]. Each technique is a (k,n) *threshold scheme*: The storage system transforms a file into n distinct blocks. A client or attacker must retrieve at least k of the n blocks to reconstruct the file. With fewer than k blocks, the client or attacker gets no information. Several of the above-mentioned systems [9, 30, 31] use these techniques to achieve security by storing each of the n pieces at a different site, and assuming that an attacker will not be able to authenticate himself to at least k of them. This avoids encryption strategies which require the secure storage of encryption keys, a difficult and dangerous practice (see [30] for a thorough discussion of this problem).

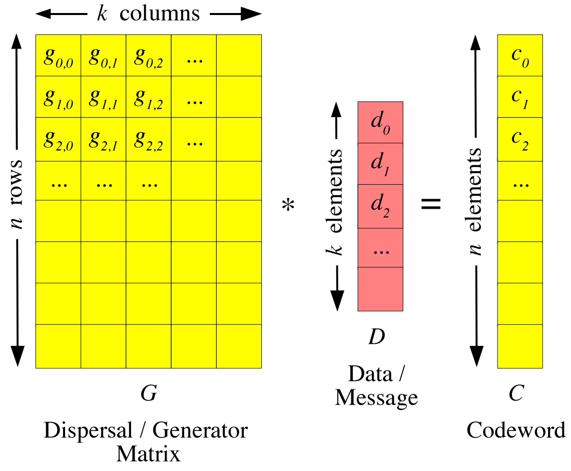
Each technique achieves a different level of security with different performance and storage requirements. If the original file is b bytes in size, Shamir's scheme requires a total of nb bytes, while Rabin's requires $\frac{nb}{k}$. Shamir's requires more computation as well. To compensate for the extra storage and computation, Shamir's scheme is more secure, achieving information theoretic security. Rabin's security is far less, and would be unacceptable in many environments.

In this paper, we describe a further modification to Rabin's scheme that achieves improved computational performance, security and integrity. We achieve this by combining the All-Or-Nothing Transform (AONT) [24] with systematic Reed-Solomon erasure codes [13].

Hence, we call it *AONT-RS*. We describe the technique, evaluate it both theoretically and experimentally and detail how it fits into a commercial dispersed storage system. We conclude with some field data of actual deployments.

2 Dispersal Algorithms

At the heart of all (k,n) threshold schemes (which we heretofore call *dispersal algorithms*) is a matrix-vector product, illustrated in Figure 1. The data to be stored is broken into *words* or *elements* which are w bits in length. A *generator* or *dispersal* matrix G is created, which has n rows and k columns. This matrix is multiplied by a k-element vector D (called the *data* or *message*) to yield a n-element vector C called the *codeword*. Each element of the codeword is stored on a different storage node.



k columns
$g_{0,0}$ $g_{0,1}$ $g_{0,2}$ ...
$g_{1,0}$ $g_{1,1}$ $g_{1,2}$ ...
$g_{2,0}$ $g_{2,1}$ $g_{2,2}$ ...
... ...
n rows
*
d_0
d_1
d_2
...
k elements
=
n elements
c_0
c_1
c_2
...
G
Dispersal / Generator Matrix
D
Data / Message
C
Codeword

Figure 1: The central matrix-vector product for all dispersal algorithms.

The dispersal matrix is constructed so that all combinations of k rows yield invertible matrices. This gives us a technique to reconstruct D from any k surviving elements of the codeword: each row of G corresponds to the calculation of a codeword element. We create a new $k \times k$ matrix A from the rows of G that correspond to the k surviving elements. We invert A and multiply A^{-1} by the surviving elements to yield D. The construction of G guarantees that A is invertible.

So that elements may fit into computer words, it is convenient that w be a power of two. To achieve this, we employ *Galois Field* arithmetic, $GF(2^w)$, where addition is equal to bitwise exclusive-or (XOR) and multiplication is implemented in a variety of ways either in hardware or software. In this way, dispersal is simply a variant of the well known Reed-Solomon codes [13, 22]. A tutorial on implementing Reed-Solomon codes in this manner is available in [17], and a thorough discussion of implementing Galois Field arithmetic is provided in [10]. There is also a methodology that converts multiplications into XOR's described in [3]. There are open-source implementations of these codes and methodologies in [16, 20, 25, 36].

Shamir's secret sharing algorithm encodes w bits of data in d_0. The remaining elements of D are randomly chosen w-bit words. The matrix G is a Vandermonde matrix, where $g_{i,j} = i^j$, which guarantees that any k rows are invertible so long as $n \leq 2^w$ [28]. Thus, when one uses Shamir's algorithm on a b-byte file, the total storage requirement is nb bytes, and the act of encoding requires $O(knb)$ XOR and multiplication operations (we will characterize this further in Section 6 below). The security guarantees of Shamir's algorithm are very strong — even with an infinite amount of computing power, unless an attacker has possession of k words, he cannot determine anything about the initial data. Moreover, this is done without the necessity of storing encryption keys.

Rabin's information dispersal algorithm (IDA) weakens the security, but improves both storage efficiency and performance. Each element of D now contains a word of data. Thus the storage requirement is $\frac{nb}{k}$ bytes, improving both storage efficiency and encoding performance by a factor of k. Like Shamir, k elements of the codeword are required to reconstruct the original data. However, the security guarantees of Rabin are far less than Shamir. We will analyze this below in Section 5, but attackers looking for known or patterned data can find it more easily from elements of the codeword. To combat this problem, Rabin suggests a technique to generate the rows of G randomly, embed the row id's within each codeword element, then encrypt the codewords [21]. Unfortunately, this requires storing an external encryption key, which does not solve the main problem we wish to solve (providing security without securely storing encryption keys).

In 1993, Krawczyk proposed a blending of Rabin and Shamir, by encrypting the data with a key-based encryption algorithm, and then dispersing the encrypted data with an IDA and the key with a secret sharing scheme [12]. This is called *Secret Sharing Made Short (SSMS)*. Our dispersal algorithm, described in the next section, also enriches Rabin's IDA with security. Unlike SSMS, it does so without secret sharing, and with the integration of integrity checking for corruption.

3 A New Dispersal Algorithm: AONT-RS

We enrich Rabin's IDA in two ways. First, we employ a variant of Rivest's *All-or-nothing Transform (AONT)* as a preprocessing pass over the data [24]. The AONT may be viewed as a $(s+1, s+1)$ threshold scheme. Data composed of s words of size w_A[1] is encoded into $s+1$ different words so that *none* of the original words may be decoded unless all $s+1$ encoded words are present, or an attacker possesses enough computing power to crack an encryption key. The key, however, is encoded with the data. If a file's size is b bytes, the performance of encoding is $O(b)$. The benefits of the AONT are:

- No external keys are necessary.
- Very little extra storage is required.
- The computational requirements of the attacker may be a parameter of the encoding.
- The performance is good.

The AONT works as follows. The data is composed of s words $d_0, \ldots, d_{s-1}$, each of which is w_A bits in length. A random key K is chosen, and each codeword c_i is calculated as:

$$c_i = d_i \oplus E(K, i+1),$$

where E is a key-based encryption algorithm such as AES [6]. A final codeword, c_k, is calculated to be a function of K and a hash of the other codewords. The AONT has *computational security,* which means that unless an attacker possesses all $s+1$ codewords or can guess K, the attacker cannot get information about *any* word or data. We will discuss this further in section 5 below.

We modify this scheme slightly. We add an extra word of data d_s, called a *canary* [2]. This word has a known, fixed value, which allows us to check the integrity of the data when it is decoded.

We generate $c_0, \ldots, c_s$ as described above and then calculate a hash h of the $s+1$ codewords using a standard hash algorithm such as SHA-256 [15] having an output at least as long as K. We then calculate a final block c_{s+1} as:

$$c_{s+1} = K \oplus h.$$

Our second modification of Rabin's IDA is to employ a *systematic* erasure code instead of a *non-systematic* one. A systematic code is defined to be one where the codeword contains the original elements of D. Without loss of generality, the first k elements of C are equal to the elements of D: $c_i = d_i$ for $0 \leq i < k$. This means that the first k rows of G compose a $k \times k$ identity matrix as pictured in Figure 2.

Employing a systematic erasure code instead of a non-systematic one (as in both the Shamir and Rabin algorithms) improves performance because it eliminates the need to encode the first k codewords. Since many systems use values of k that are large relative to n (e.g. POTSHARDS' evaluation uses a (3,5) Shamir scheme [30]) the savings during encoding with a systematic erasure code are substantial. Moreover, when decoding, codeword elements that are equal to data elements do not have to be decoded, which improves performance further.

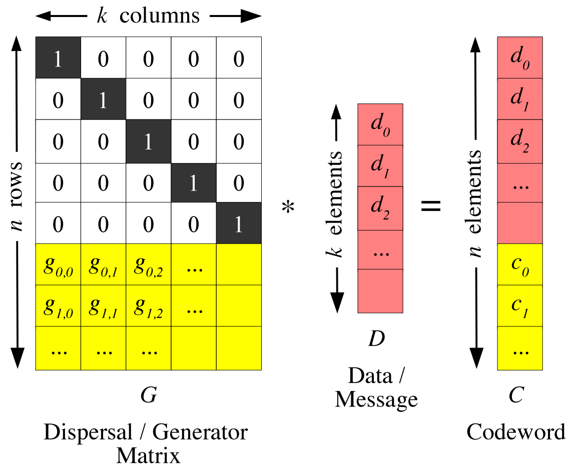


Figure 2: A systematic erasure code.

We call our dispersal technique *AONT-RS*, as it is a combination of the All-Or-Nothing Transform and Reed-Solomon coding. The intuition is that we use the AONT for security and the dispersal for availability, proximity and fault-tolerance. This is unlike Shamir, Rabin and SSMS which use dispersal to achieve both functions.

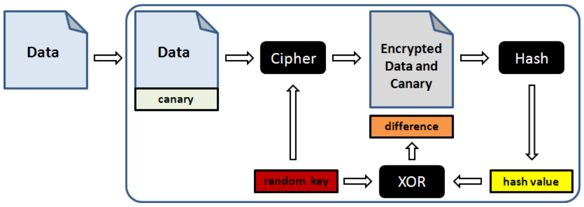


Figure 3: Encoding operation of AONT.

Several diagrams depict the operation of AONT-RS and interaction between AONT and Reed-Solomon coding. In Figure 3, data is processed by AONT. A canary is appended to the data, and the data and canary are encrypted with a random key. A hash value of the encrypted data is computed. The hash value and random key are then combined via bitwise exclusive-or to form a difference, which is appended to the encrypted data to form the AONT package.

Once processed by AONT, the result is treated as normal input to a systematic IDA, as depicted in Figure 4.

[1] Since AONT-RS mixes AONT with dispersal, we differentiate its word size from the dispersal's word size using w_A instead of w.

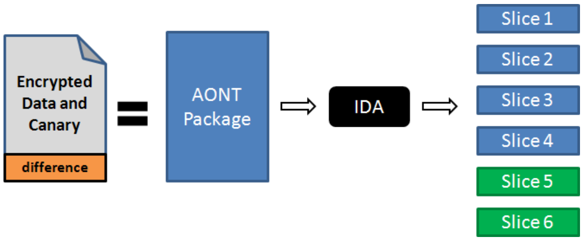


Figure 4: Dispersal of AONT package using a systematic IDA such as Reed-Solomon coding.

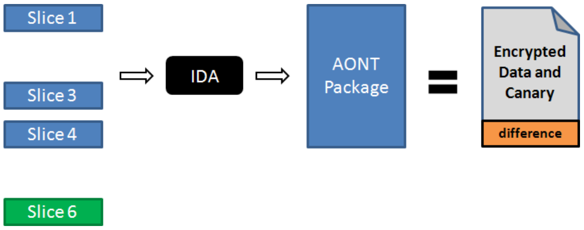


Figure 5: Recovering the AONT package from a threshold number of slices.

The IDA splits the input into k slices formed directly from the input and computes $n-k$ coding slices. Slices are then stored to separate locations.

At a future time, slices may be retrieved and used to recover the data. The first step in this process requires obtaining a threshold number of slices, as in Figure 5. Short of a threshold number of slices the entire AONT package cannot be recovered; there is not enough information contained in $m<k$ slices to yield the original input, whose length is k times the slice length. However, if one possesses any k of the slices, they may compute the original input to the IDA which in this case is the AONT package.

As shown in Figure 6, Reversing the AONT operation is trivial when one possesses the entire package. The first step is to compute the hash, h, of the encrypted data. Since the last block contains $K\oplus h$ and we know the hash value h, we may exclusive-or the last block with the hash to find $(K\oplus h\oplus h)$. Since $h\oplus h$ equals zero, the result is the random key K. The random key is then used to decrypt the encrypted data, and the canary is checked to detect corruption.

4 A Concrete Example

To help illustrate, we present a concrete example. Suppose we have a 4KB block of data, D that we wish to massage into 16 *slices* on 16 storage nodes so that we may reconstruct and verify the data so long as we possess any 10 slices.

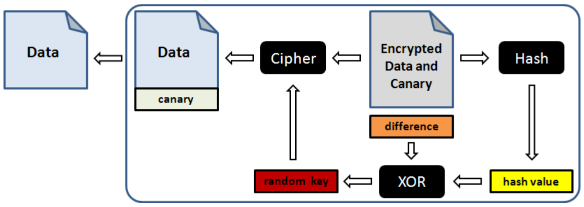


Figure 6: Restoring data from an AONT package.

Shamir: To apply Shamir's algorithm, we view the data as 4096 individual bytes, $d_0,\ldots,d_{4095}$. Each of the 16 slices $S_0,\ldots S_{15}$ will also be composed of 4096 individual bytes $s_{i,0},\ldots,s_{i,4095}$ such that $s_{i,j}$ is a function of d_j and nine random bytes. Specifically,

$$s_{i,j}=d_j\oplus\sum_{x=1}^{9}(i+1)^x r_{j,x},$$

where $r_{j,x}$ is a random byte and arithmetic is over $GF(2^8)$. The total storage requirement is 64 KB.

Rabin: To apply Rabin, we pad D to be 4100 bytes and then partition it into ten data slices $DS_0,\ldots,DS_9$ of 410 bytes each. As with Shamir, we view each data slice DS_i to be composed of 410 individual bytes $DS_{i,0},\ldots Ds_{i,409}$. We then calculate each of the 16 slices using Reed-Solomon coding on the individual bytes: [2]

$$s_{i,j}=\sum_{x=0}^{9}(i+1)^x d_{x,j}.$$

Again, arithmetic is over $GF(2^8)$. The total storage requirement is 16*410 = 6.41 KB.

SSMS: With SSMS, we select a random 16-byte encryption key and encrypt the data with an encryption algorithm such as AES. We then disperse it using Rabin and disperse the key using Shamir. The total storage requirement is 16*(410+16) = 6.65 KB.

AONT-RS: We will be adding 34 additional bytes to the data, and we will first view it as being composed of 257 16-byte words, $d_0,\ldots,d_{256}$, where the first 256 words are the original data. We set d_{256} to be a 16-byte canary value. We choose K to be sixteen random bytes and set each c_i to equal $d_i\oplus E(K,i+1)$ where E is a standard encryption algorithm. Next we calculate h to be a 16-byte hash of $c_0,\ldots,c_{256}$. Finally, we set c_{257} to equal $h\oplus K$. The last 2 bytes are immaterial – they are simply padding so that the data may be partitioned into ten equal slices. They could be used as additional canaries if desired.

[2] While Rabin does not use a Vandermonde matrix in [21], the matrix he employs has the same properties.

As with Rabin, we partition the 4130 bytes into ten data slices $DS_0, \ldots, DS_9$ of 413 bytes each. These will be stored on the first ten storage nodes. Six additional coding slices $CS_0, \ldots, CS_5$ will be calculated using a different dispersal matrix, such as the one depicted in Figure 7, which is derived from the Vandermonde matrix for systematic coding (see [18] for an explanation of why a Vandermonde matrix is inadequate for this purpose). The total storage requirement is 16*413 = 6.45 KB.

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 147 & 138 & 73 & 93 & 161 & 103 & 58 & 99 & 178 \\ 1 & 103 & 156 & 151 & 123 & 187 & 166 & 175 & 244 & 83 \\ 1 & 58 & 203 & 60 & 48 & 51 & 175 & 52 & 16 & 30 \\ 1 & 93 & 151 & 205 & 212 & 44 & 123 & 48 & 197 & 244 \\ 1 & 220 & 166 & 123 & 82 & 143 & 245 & 40 & 167 & 122 \end{pmatrix}$$

Figure 7: Dispersal matrix for the systematic $(10,16)$ Reed-Solomon code over $GF(2^8)$.

In each of the four methods, a client or attacker needs to acquire 10 of the 16 slices to read the data. Each method has different security and performance characteristics, which are included in the sections of Security and Performance below.

5 Security Evaluation

The threat model that we use is one where individual storage servers belong to different domains, both administrative and physical. Servers may be lost due to non-security-related events like power failure or water damage, or their security may be compromised; for example a rogue system administrator or outside attacker can steal data. Moreover, servers may become corrupted either maliciously or due to the natural process of time. We assume that the physical dispersal of storage servers is limiting on an attacker, and that the difficulty of breaching servers in multiple domains, along with a judicious choice of k and n, is sufficient to make the system secure.

All of these schemes provide a good level of security – if one cannot truly decode the data without acquiring all k slices, then an attacker without some *a priori* information about the data will not be able to glean anything from fewer than k slices. In the words of Rabin, "We do not see a way of fully reconstructing even small portions of D from $k-1$ pieces" [21]. [3]

However, if an attacker has some notion of what data he or she is seeking but possesses fewer than $k-1$ slices, then the schemes differ greatly. We will consider the most pathological example: An attacker possesses $m < k$ slices of the codeword C and wants to verify whether the data that it encodes matches some predetermined value. Further, if the attacker can verify that one slice of D matches, then the attacker can be assured that the rest matches. While this seems rather generous to the attacker, there are many realistic attacking scenarios that can be reduced to this one [7]. For each algorithm, we assume that the attacker knows how the slices were generated, except for the random numbers.

Shamir: Shamir's security is guaranteed. Attackers cannot get any information from fewer than k slices, regardless of their computing power. For example, with $k-1$ slices each of size w, there are 2^w potential values of d_0 that can generate those slices. Thus, every possible value of d_0 is equally likely. One needs the k-th slice to determine the actual value of d_0. This is *information theoretic* security.

Rabin: Since Rabin's IDA has no randomness, it has no security, even if the attacker owns just one slice. Since the attacker knows how the slices are generated, compromise consists solely of verifying that a slice has a predetermined value. Further, if the generator matrix is known and the data has recognizable patterns (i.e. it is not random looking) then it is possible to guess the content of missing slices. If one has $k-1$ slices, trying each of the 2^w possibilities for words of a missing slice will yield k recognizable words when the correct value is attempted.

SSMS: SSMS has *computational security* [12]. Without the key, one has to break the encryption, which can be made computationally intractible with a large enough key. Moreover, since Shamir protects the key with information theoretic security, there is no way get the key with fewer than k slices.

AONT-RS: AONT has the property that unless one has all of the encrypted data, one cannot decode any of it. This is because one needs all of the data to discover K, and one cannot decode any of the data without K. However, if an attacker owns K and one slice, then the attacker can easily verify that D has a predetermined value, just as in Rabin. Thus, we analyze the difficulty in having the attacker figure out K's value. Suppose the attacker owns the first slice, which contains the first encoded word of D, which is equal to $d_0 \oplus E(K,1)$. The encoding function guarantees that enumeration is the only way to discover K's value, which means that an attacker must test up to 2^{w_A} potential values of K to discover its real value. Like SSMS, this is *computational security*.

Thus, both AONT-RS and SSMS have computational security. If an attacker owns any data slice, then compromise can only occur by discovering K as above. If an attacker owns a coding slice, then the attacker must again enumerate potential values of K, calculate potential values of the slice and verify them. Owning $k-1$

[3] We have changed the variables in the quote to match our paper.

Algorithm	Running Time	Storage
Shamir	$\mathbf{Perf}(n,k,kb)$	nb
Rabin	$\mathbf{Perf}(n,k,b)$	$\frac{nb}{k}$
AONT-RS	$\mathbf{AONT}(b)+\mathbf{Perf}(n-k,k,b)$	$\frac{n(b+w_A)}{k}$

Table 1: Running time and storage requirements of the three dispersal algorithms.

slices adds no information – the act of verification still boils down to enumerating all potential values of K. The encryption and therefore missing words in other slices cannot be guessed in the same way they can under Rabin.

Special mention must be made of storing $K \oplus h$ as the last element of the codeword. Cryptographic hash functions are designed to have an unpredictable and uniformly distributed output. Further, they are designed to follow the strict avalanche criterion [35], meaning h is dependent on every bit of input. Therefore unless an attacker knows all code words $c_0,\ldots,c_s$, h cannot be predicted. Modeling the hash function as a random oracle, h encrypts K in the same manner as a One-Time-Pad [34] and provides information theoretic security since h is the same length as K. Therefore $K \oplus h$ yields no information about K when h is unknown.

Moreover, the avalance criterion allows the canary to be sufficient to check integrity. If any bit of the stored slices is modified, then with sufficient probability, the calculated hash h' will be different from the one used to calculate the difference. Since h' differs from h, the calculated encryption key K will be incorrect, and as a result, the value in the calculated canary will differ from its known value.

While computational security is not as strong as information theoretic security, in our view it is functionally equivalent. As long as w_A is sufficiently large, it is computationally infeasible for an attacker to even verify that slices hold given data. For example, when $w = 256$ as in Section 4, compromise requires the enumeration of 2^{256} keys. To put this in perspective, if each person on earth had access to a trillion computers that can test a trillion keys per second, it would take over 10^{35} years on average to correctly guess the key. According to some estimates of proton half-life, most matter in the universe will have decayed before the key would be found [1].

6 Theoretical Performance

Let $\mathbf{Perf}(R,C,S)$ be the CPU time that it takes to encode D, composed of S total bytes, with a $R \times C$ dispersal matrix. In terms of big-O notation, $\mathbf{Perf}(R,C,S) = O(RCS)$. A more precise evaluation of $\mathbf{Perf}(R,C,S)$ is difficult, because of the variety of ways that the encoding may be implemented. If one implements the encoding with standard finite field arithmetic, then:

$$\mathbf{Perf}(R,C,S) = \frac{S}{C}\left(\frac{(R-1)(C-1)}{\mathbf{Mult}} + \frac{R(C-1)}{\mathbf{XOR}}\right),$$

where **Mult** is the bandwidth of performing Galois Field multiplication and **XOR** is the bandwidth of performing XOR operations. This is because encoding becomes a series of dot products to create R coding slices each of whose size is $\frac{S}{C}$ bytes. The difference in the number of multiplications vs. XORs arises becuase nearly all dispersal matrices are like Figure 7 and have ones in their top rows and leftmost columns. Implementations of Reed-Solomon coding do, however, differ in their performance characteristics. Using Cauchy Reed-Solomon coding [3], for example, substitutes additional XOR operations for the multiplication and can improve performance significantly [19].

Additionally, let $\mathbf{AONT}(S)$ be the time that it takes to perform the AONT on S bytes of data. The choice of w_A, encryption and hashing technique will all affect $\mathbf{AONT}(S)$. In general, though, it is $O(S)$ and is also easy to parallelize [24].

Given the parameters k, n, b, $\mathbf{Perf}(R,C,S)$, and $\mathbf{AONT}(S)$ the performance of the three main dispersal algorithms and their storage requirements are given in Table 1. Since SSMS doesn't specify a recommended dispersal or encryption algorithm, we omit it from the remaining analyses. Roughly, its performance will be close to AONT-RS.

7 Microbenchmark Performance

To assess actual performance, we used open-source C libraries to perform the various functionalities. All tests were performed on a 4-core Intel Xeon W3530 at 2.80 GHz with 6 GB of memory at 1066 MHz running Linux kernel 2.6.32. Despite having multiple cores, all benchmarks were performed using a single thread. For Reed-Solomon coding, we used Luigi Rizzo's open source library over $GF(2^8)$ [25]. We tested a variety of k-of-n configurations, ranging from 3-of-6 to 32-of-64, measuring c_e, defined as the bandwidth of creating each coding slice, times k. For a given machine, c_e should be relatively constant, since the time to create each coding slice

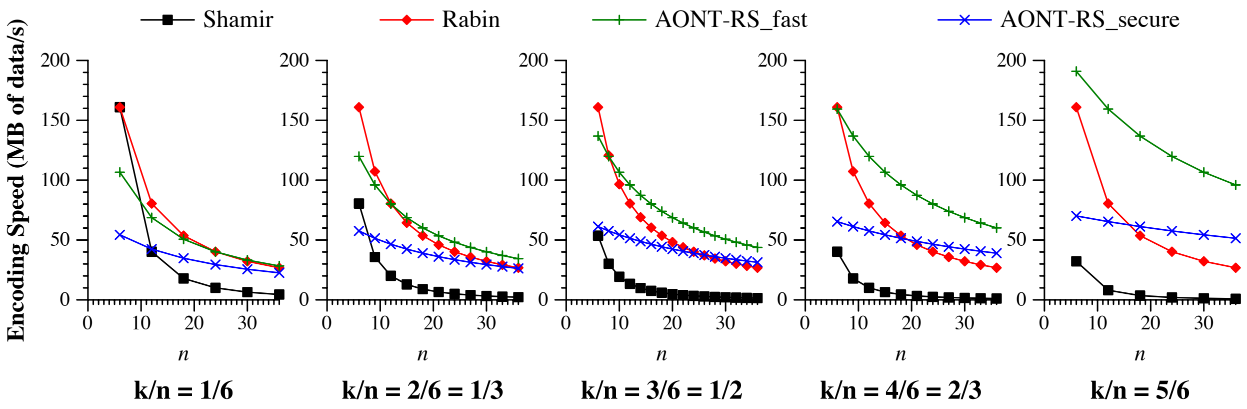


Figure 8: Performance comparison of the dispersal algorithms. Each graph affixes the k-to-n rate and plots speed of encoding with each dispersal algorithm.

should be linear in k. Despite the wide disparity in configurations, we observe that c_e is fairly consistent, with a minimum of 921.60 MB/s in the 3-of-6 configuration, to a maximum of 994.00 MB/s in 27-of-54. The average performance for the 30 configurations tested is 965.61 MB/s with a standard deviation of 11.42 MB/s. Thus, we can use c_e to approximate **Perf** as:

$$\textbf{Perf}(R,C,S) = \frac{RCS}{965.61MB/s}.$$

The encoding time for AONT is dependent on the choice of cipher and hash function. To encode S bytes using AONT, both the cipher and hash function must process S bytes. Therefore the time equals the sum of the time to encrypt S bytes plus the time to hash S bytes. We tested the performance of two pairs of cipher/hash algorithms, one tailored for high security (AES-256 and SHA-256) and the other tailored for performance (RC4-128 and MD5). For this test, we used OpenSSL 0.9.8k with a block size of 8 KB. The results are in Table 2.

	Encoding Rate (MB/s)
AES-256	143.30
RC4-128	414.17
SHA-256	160.03
MD5	559.47

Table 2: Performance of two encryption algorithms (AES-256 and RC4-128) and two hash algorithms (SHA-256 and MD5).

Thus, we come up with two functions for $\textbf{AONT}(S)$, one which we call **secure** (AES-256 and SHA-256), and one which we call **fast** (RC4-128 and MD5):

$$\textbf{AONT}_{\textbf{secure}}(S) = \frac{S}{75.60MB/s}$$

$$\textbf{AONT}_{\textbf{fast}}(S) = \frac{S}{237.99MB/s}$$

We now have the necessary information to use Table 1 to evaluate the performance of the three dispersal algorithms for any k-of-n configuration. We do so in Figure 8. Each graph affixes a k-of-n ratio called a *rate* and then plots the speed of encoding in MB of data per second. The rates increase by $\frac{1}{6}$ for each successive graph, starting with a very low rate of $\frac{1}{6}$ and proceeding to a very high rate of $\frac{5}{6}$.

The trade-offs of the various formulas are apparent from the graph. There is a dispersal cost for all three algorithms and an AONT cost for the AONT-RS algorithms. The AONT cost is constant, since it depends solely on the size of the data. Thus, when dispersal is very fast, as in the 1-of-6 and 2-of-6 cases, Rabin outperforms AONT-RS$_{\textbf{fast}}$ and Shamir outperforms AONT-RS$_{\textbf{secure}}$. As k and n grow, however, the dispersal costs increase. This increase is most pronounced with Shamir, then with Rabin and finally with AONT-RS. For each rate except the very low $\frac{1}{6}$, there is a point where the performance of AONT-RS$_{\textbf{fast}}$ becomes the best, and a point where AONT-RS$_{\textbf{secure}}$'s performance surpasses both Shamir and Rabin. These points come at lower values of n for higher k-of-n rates.

A schematic of Cleversafe's storage architecture is depicted in Figure 9. Although not plotted above, of special interest is the 3-of-5 data point, since this is the k-of-n configuration measured by POTSHARDS [30], an archival storage system that uses Shamir for both fault-tolerance and security. For this configuration, the perfor-

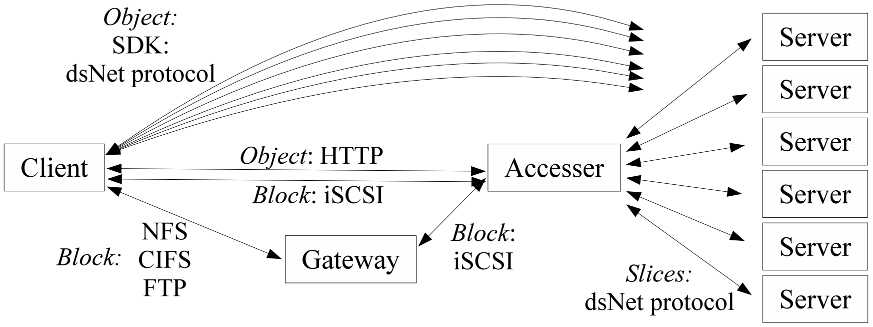


Figure 9: A high-level picture of Cleversafe's storage architecture.

mance of AONT-RS$_{\textbf{secure}}$ (65.4 MB/s) is nearly identical to Shamir (64.4 MB/s), which means that a system like POTSHARDS can achieve computational security rather than information theoretic security for the same performance, but with a factor of three less storage.

8 Commercial Dispersed Storage System

AONT-RS is a feature in the storage software and appliances sold by Cleversafe, which developed the technique to address the threat model of compromise, theft or loss of disks and devices. By appropriately tuning the dispersal configuration, all disks or devices at an entire site can be stolen and the data will remain confidential. Similarly, as long as a minimum threshold of servers are available, subsets of servers may be brought offline temporarily for maintenance, or permanently for replacement. Since the servers are protected by AONT-RS, storage owners may dispose of servers without having to "wipe" the drives clean, since the information on the servers is impossible to obtain without gaining access to some subset of the remaining servers.

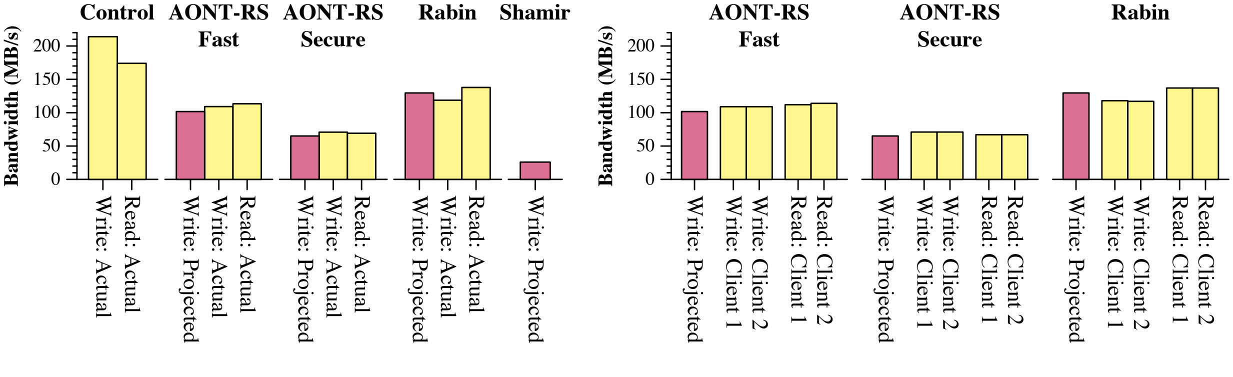
Two paradigms are exposed to clients — a block paradigm that supports standard protocols like NFS, CIFS, FTP and iSCSI, and an object paradigm that supports larger storage units for better performance. An *Accesser* calculates mappings that associate blocks or objects to slices on dispersed storage servers (termed "Slicestores" in Cleversafe's product). A common configuration is to encode each block or object into 16 slices using a $(10,16)$-threshold AONT-RS scheme.

Block reads and writes that use iSCSI go through the Accesser. The Accesser performs the block-to-slice encoding and decoding, and also manages the traffic to and from the servers. The other protocols require a *Gateway*, typically co-located with the Accesser, that translates between the various file protocols and iSCSI. Since this path has two hops and interacts with the servers with small messages, the performance of the block protocols is limited by the networking hardware and not the AONT-RS protocol. Storage servers do support multiple Accessers, which relieves one bottleneck of the block-based system.

To achieve better performance, Cleversafe also exports a protocol for large objects. Objects are partitioned into Megabyte-sized chunks, which are then encoded into slices for dispersal. Clients may either read and write objects through the Accesser using HTTP, or they may use a SDK to perform their own AONT-RS encoding/decoding so that they may interact directly with the servers. In both cases, the client manages the context of the object name. A common software architecture is that clients use a database to maintain the the meaning and relationships of the content, and they store the object names in a column of the database.

Slice pointers are 48 bytes in length and are composed of three parts: *routing information* that enables slices to be routed to and from the correct servers, the *source name* which identifies the slice, and *vault information* which enables access control. The source name is opaque – its interpretation is dependent on the specific client and server. Vaults are logical containers of storage. Each vault has its own quotas, data coding parameters and access controls. Access controls are identity-based; each vault may have an arbitrary number of accounts granted read or write permissions to it.

Each slice is stored with metadata that identifies the slice's coding parameters and a version number. The version number is increased for each distinct write of the block or object, and concurrency control is maintained via the SDK with transactions and a three-phase commit. An additional parameter of each system is the *write threshold*, z, where $k \le z \le n$. This specifies how many slices must be written before a write can be committed. Setting z closer to k improves latency at the expense of reliability for a window of time. The remaining $(n-z)$ writes are processed in the background, which reduces this window of exposure.



(a) One client

(b) Two clients

Figure 10: Actual and projected performance of dispersed storage of 10 MB objects on a $(5,8)$ test configuration.

Authentication in the system is two-way: servers authenticate themselves to clients by means of a digital certificate, which identifies it within the dispersed storage system and allows TLS sessions to be created. The method of authentication of the client to the server is flexible — both password and certificate-based authentication are supported. Despite use of AONT-RS, secure network communication is still required for security since a threshold number of slices travel together over the 'last mile' of the client's connection.

All components are written in Java. Reed-Solomon erasure coding is performed using Java's FEC library [16], and encryption using SunJCE.

9 Measured Performance

To measure performance, we use a commercial configuration with one or two clients and eight servers. The client and Accesser machines each have a 4-core Intel(R) Xeon(R) X3430 processor running at 2.40 GHz with 8 MB cache and 16 GB of ECC RAM. Four GB of memory is allocated to the JVM when executing the software. We use the Java HotSpot(TM) 64-Bit Server VM (build 17.0-b16, mixed mode) running Java 1.6.0_21. The storage servers each have a 4-core Intel(R) Xeon(R) X3460 processor at 2.80 GHz with 8 MB cache and 16 GB of ECC RAM. For storage, each server has twelve 2 TB Seagate SATA drives. The networking between components consists of a 10 Gb Ethernet switch. To handle simultaneous connections to multiple servers, the Accessers have 10 Gb network interface cards. The servers' cards are 1 Gb.

Our main test has the client spend 10 minutes reading and writing 10 MB objects, held in main memory, to the eight-server storage network, using the SDK and object interface. The coding parameters are $k = 5$ and $n = 8$, and five threads are employed by the client to leverage all of its cores. As in section 7, we recorded microbenchmarks of the various components of dispersal:

$$\begin{aligned}\textbf{AONT}_{\textbf{secure}}(S) &= \frac{S}{104.77MB/s}\\ \textbf{AONT}_{\textbf{fast}}(S) &= \frac{S}{249.03MB/s}\\ c_e &= 2628MB/s\end{aligned}$$

The performance of a control and the dispersal algorithms is shown in Figure 10(a). The control has the client perform no encoding, but still sends 8 slices to the servers. While the Cleversafe implementation is flexible, allowing us to embed Rabin and both AONT-RS dispersal algorithms, we did not implement Shamir within the framework. This is because the blowup of storage requirements by a factor of five would be unreasonable.

We show the actual performance of writes and reads for the control, the two AONT-RS implementations and Rabin. We also include the projected write performance of the dispersal algorithms, including Shamir, using the performance equations from section 7, the microbenchmarks, plus the performance of the control as the actual dispersal bandwidth (214 MB/s).

For the three dispersal algorithms that we tested, the projected performance was within ten percent of the actual performance. We find this result compelling because the system on which the tests were performed was a production-level system, implementing the full functionality of Cleversafe's commercial storage system, including access control and metadata management.

In the tests with coding, the CPU utilization of the client is measured to be 90%. Since the closest I/O bottlenecks are the eight 1-Gbps links to the storage servers, it is clear that the limiting factor in these tests is the ability of the client computer to process data. To further affirm the client as bottleneck, we ran two clients simultaneously and present their performance in Figure 10(b). The clients' performance is nearly identical to Figure 10(a).

It is worth noting that AONT-RS$_{\textbf{secure}}$ exhibits worse performance when reading than when writing; we expected that during reads, less CPU resources would be required, since some slices do not need to be processed by the IDA. The worse performance is due to the SunJCE implementation of AES, which is significantly slower when decrypting than when encrypting. In a stand-alone benchmark we observed 31.51 MB/s vs. 44.77 MB/s when encrypting.

10 Tales of Deployment

Today, there are over 20 Cleversafe dispersed storage installations in pilot and production around the world, with customers drawing from a diverse set of industries including financial, health care, entertainment, and defense. Several customers (who have asked to remain anonymous) have cited one important factor in their purchasing decision: that the contents of small sets of servers are meaningless in isolation. Thus, one can decommission disk drives or potentially even server sites without having to "wipe" them, which can be expensive [4]. Since nearly all U.S. states have "data breach laws," that require companies to proactively disclose the loss of storage that is not encrypted [33], using AONT-RS can save companies time, attorney fees and bad publicity that results from having to alert consumers to a data breach.

One of Cleversafe's deployments is for The Museum of Broadcast Communications that serves its video collections on the Internet. In particular, over 8,500 hours of historical audio and video content have been digitized and stored on tens of terabytes in one of Cleversafe's dispersed storage systems. Roughly 200,000 monthly visitors access the archives over the web.

The Museum deployment is composed of 16 storage servers, each having 4 TB of raw capacity and spread across 8 sites: Chicago, Dallas (two locations) Denver, New Jersey, San Francisco, Seattle and Tampa. The sites are situated across three power grids in the continental United States, and the data is dispersed in a 10-of-16 configuration. In this way, even if one entire power grid shuts down, enough servers will remain accessible to retrieve all the data. The Museum uses the object store interface inside its internal database, so that users employ the database to search a rich set of metadata about the movies, which can then be retrieved using the object handle.

Internally, Cleversafe maintains dispersed storage systems having over 1 PB of capacity. These are used internally for development, testing and storing production data. Employees have their own personal vaults with access to a 30 TB pool of dispersed storage, which is implemented over 8 geographically separated storage servers across the United States.

In one case, Cleversafe initially deployed a system across four sites, but at a later time decided that it should be migrated to 8 sites to provide better tolerance to site and power grid outages. To accomplish this without bringing the system down, machines were incrementally boxed up and shipped across the country, such that at all times a threshold number remained online. Therefore the system remained accessible for reads and writes throughout the process. The same essential technique is now used to apply software updates. Nodes are upgraded individually allowing the system to maintain availability throughout the upgrade process.

11 Conclusion

Dispersed storage systems enable availability, scalability, and performance based on physical proximity. They also enable security via (k,n) threshold schemes that require attackers to authenticate themselves to k of n storage nodes in order to read data. The threshold schemes provide this security without relying on the secure storage of encryption keys, which is a notoriously difficult problem.

We have described a new dispersal algorithm called AONT-RS, which combines the All-Or-Nothing Transform with systematic Reed-Solomon codes to achieve computational security. Compared to traditional approaches to dispersal, AONT-RS has a very attractive blend of properties. Its storage and computational footprint is much less than Shamir secret sharing. While Shamir achieves information theoretic security AONT-RS's security can be tuned so that compromise is computationally infeasible. Compared to Rabin's classic dispersal algorithm, AONT-RS achieves a far greater degree of security, and also better performance for larger installations. This is because AONT-RS is based on a systematic Reed-Solomon erasure code rather than the non-systematic code employed by Rabin. We have detailed the theoretical and applied performance of the dispersal algorithms, and described a commercial dispersed storage product that is based upon the dispersal algorithm.

AONT-RS is not specific to our dispersal solution. For example, the POTSHARDS archival storage system [30] could use AONT-RS to implement computational rather than information theoretic security and reduce their storage requirements by a factor of three. Other solutions such as Gridsharing [31] can improve their security by

[4] For example, see `http://www.east-tec.com/enterprise/disposesecureent/`.

employing AONT-RS rather than a standard systematic Reed-Solomon code.

In future work, we would like to collect data from our private and commercial deployments concerning failures, node availability, compromise and attack. Such data will enable us to make better policy decisions concerning configurations of dispersed storage. These decisions will allow us to tune the AONT and erasure code configuration used, and will also allow us to make the most efficient use of our storage.

12 Acknowledgements

This material is based upon work supported by the National Science Foundation under grants CNS-0615221 and CSR-1016636. The authors gratefully acknowledge Ilya Volvovski's contribution to the work, plus Steve Hand for shepherding the paper through the final reviewing process.

References

[1] AMSLER *et al*, C. Review of particle physics. *Physics Letters B 667*, 1 (2008).

[2] AYCOCK, J. *Computer Viruses and Malware (Advances in Information Security)*. Springer-Verlag, New York, 2006.

[3] BLOMER, J., KALFANE, M., KARPINSKI, M., KARP, R., LUBY, M., AND ZUCKERMAN, D. An XOR-based erasure-resilient coding scheme. Tech. Rep. TR-95-048, International Computer Science Institute, August 1995.

[4] CLEVERSAFE, INC. Cleversafe dispersed storage. Community portal:`www.cleversafe.org`, 2010.

[5] COHEN, B. Incentives build robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer Systems* (Berkely, CA, June 2003).

[6] DAEMEN, J., AND RIJMEN, V. *The Design of Rijndael, AES — The Advanced Encryption Standard*. Springer-Verlag, New York, 2002.

[7] FERGUSON, N., SCHNEIER, B., AND KOHNO, T. *Cryptography Engineering*. John Wiley & Sons Ltd, Chichester, 2010.

[8] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S. T. The Google file system. In *19th ACM Symposium on Operating Systems Principles (SOSP '03)* (2003).

[9] GOODSON, G. R., WYLIE, J. J., GANGER, G. R., AND REITER, M. K. Efficient byzantine-tolerant erasure-coded storage. In *DSN-04: International Conference on Dependable Systems and Networks* (Florence, Italy, 2004), IEEE.

[10] GREENAN, K., MILLER, E., AND SCHWARTZ, T. J. Optimizing Galois Field arithmetic for diverse processor architectures and applications. In *MASCOTS 2008: 16th IEEE Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems* (Baltimore, MD, September 2008).

[11] HAEBERLEN, A., MISLOVE, A., AND DRUSCHEL, P. Glacier: Highly durable decentralized storage despite massive corrolated failures. In *2nd Symposium on Networked Systems Design and Implementation (NSDI)* (2005).

[12] KRAWCZYK, H. Secret sharing made short. In *13th Annual International Conference on Advances in Cryptology* (1993).

[13] MACWILLIAMS, F. J., AND SLOANE, N. J. A. *The Theory of Error-Correcting Codes, Part I*. North-Holland Publishing Company, Amsterdam, New York, Oxford, 1977.

[14] MANIATIS, P., ROSENTHAL, D. S. H., ROUSSOPOULOS, M., AND BAKER, M. LOCKSS: A peer-to-peer digital preservation system. *ACM Transactions on Computer Systems 23* (2003).

[15] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. Secure hash standard (shs). FIPS PUB 180-3, `http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf`, October 2008.

[16] ONION NETWORKS. Java FEC Library v1.0.3. Open source code distribution: `http://onionnetworks.com/fec/javadoc/`, 2001.

[17] PLANK, J. S. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software – Practice & Experience 27*, 9 (September 1997), 995–1012.

[18] PLANK, J. S., AND DING, Y. Note: Correction to the 1997 tutorial on Reed-Solomon coding. *Software – Practice & Experience 35*, 2 (February 2005), 189–194.

[19] PLANK, J. S., LUO, J., SCHUMAN, C. D., XU, L., AND WILCOX-O'HEARN, Z. A performance evaluation and examination of open-source erasure coding libraries for storage. In *FAST-2009: 7th Usenix Conference on File and Storage Technologies* (February 2009), pp. 253–265.

[20] PLANK, J. S., SIMMERMAN, S., AND SCHUMAN, C. D. Jerasure: A library in C/C++ facilitating erasure coding for storage applications - Version 1.2. Tech. Rep. CS-08-627, University of Tennessee, August 2008.

[21] RABIN, M. O. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the Association for Computing Machinery 36*, 2 (April 1989), 335–348.

[22] REED, I. S., AND SOLOMON, G. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics 8* (1960), 300–304.

[23] RHEA, S., WELLS, C., EATON, P., GEELS, D., ZHAO, B., WEATHERSPOON, H., AND KUBIATOWICZ, J. Maintenance-free global data storage. *IEEE Internet Computing 5*, 5 (2001), 40–49.

[24] RIVEST, R. All-or-nothing encryption and the package transform. In *4th International Workshop on Fast Software Encryption* (1997), pp. 210–218.

[25] RIZZO, L. Erasure codes based on Vandermonde matrices. Gzipped **tar** file posted at `http://planete-bcast.inrialpes.fr/rubrique.php3?id_rubrique=10`, 1998.

[26] ROWSTRON, A., AND DRUSCHEL, P. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. *ACM SIGOPS Operating Systems Review 35*, 5 (2001), 188–201.

[27] SANTRY, D. S., FEELEY, M. J., HUTCHINSON, N. C., VEITCH, A. C., CARTON, W., AND OFIR, J. The Google file system. In *17th ACM Symposium on Operating Systems Principles (SOSP '99)* (1999).

[28] SHAMIR, A. How to share a secret. *Communications of the ACM 22*, 11 (November 1979), 612–613.

[29] STORER, M. W., GREENAN, K. M., MILLER, E. L., AND VORUGANTI, K. Pergamum: Replacing tape with energy efficient, reliable, disk-based archival storage. In *FAST-2008: 6th Usenix Conference on File and Storage Technologies* (San Jose, February 2008), pp. 1–16.

[30] STORER, M. W., GREENAN, K. M., MILLER, E. L., AND VORUGANTI, K. POTSHARDS – a secure, long-term storage system. *ACM Transactions on Storage 5*, 2 (June 2009).

[31] SUBBIAH, A., AND BLOUGH, D. M. An approach for fault tolerant and secure data storage in collaborative work environments. In *ACM Workshop on Storage Security and Survivability* (2005).

[32] TAHO-LAFS. Tahoe least authority file system. Open source code distribution: `http://tahoe-lafs.org/trac/tahoe-lafs`, 2010.

[33] VANCE, K. Keeping pace with data encryption laws. `www.esecurityplanet.com/trends/article.php/3887111`, June 2010.

[34] VERNAM, G. S. Cipher printing telegraph systems for secret wire and radio telegraphic communications. *Journal of the IEEE 55* (1926), 109–115.

[35] WEBSTER, A. F., AND TAVARES, S. E. On the design of S-boxes. In *Advances in Cryptology - Crypto '85* (1985), Springer-Verlag, pp. 523–534.

[36] WILCOX-O'HEARN, Z. Zfec 1.4.0. Open source code distribution: `http://pypi.python.org/pypi/zfec`, 2008.

Emulating Goliath Storage Systems with David

Nitin Agrawal[†], *Leo Arulraj*[*], *Andrea C. Arpaci-Dusseau*[*], *Remzi H. Arpaci-Dusseau*[*]
NEC Laboratories America[†] *University of Wisconsin–Madison*[*]
nitin@nec-labs.com *{arulraj, dusseau, remzi}@cs.wisc.edu*

Abstract

Benchmarking file and storage systems on *large* file-system images is important, but difficult and often infeasible. Typically, running benchmarks on such large disk setups is a frequent source of frustration for file-system evaluators; the scale alone acts as a strong deterrent against using larger albeit realistic benchmarks. To address this problem, we develop David: a system that makes it practical to run large benchmarks using modest amount of storage or memory capacities readily available on most computers. David creates a "compressed" version of the original file-system image by omitting all file data and laying out metadata more efficiently; an online storage model determines the runtime of the benchmark workload on the original uncompressed image. David works under any file system as demonstrated in this paper with ext3 and btrfs. We find that David reduces storage requirements by orders of magnitude; David is able to emulate a 1 TB target workload using only an 80 GB available disk, while still modeling the actual runtime accurately. David can also emulate newer or faster devices, *e.g.*, we show how David can effectively emulate a multi-disk RAID using a limited amount of memory.

1 Introduction

File and storage systems are currently difficult to benchmark. Ideally, one would like to use a benchmark workload that is a realistic approximation of a known application. One would also like to run it in a configuration representative of real world scenarios, including realistic disk subsystems and file-system images.

In practice, realistic benchmarks and their realistic configurations tend to be much larger and more complex to set up than their trivial counterparts. File system traces (*e.g.*, from HP Labs [17]) are good examples of such workloads, often being large and unwieldy. Developing scalable yet practical benchmarks has long been a challenge for the storage systems community [16]. In particular, benchmarks such as GraySort [1] and SPECmail2009 [22] are compelling yet difficult to set up and use currently, requiring around 100 TB for GraySort and anywhere from 100 GB to 2 TB for SPECmail2009.

Benchmarking on large storage devices is thus a frequent source of frustration for file-system evaluators; the scale acts as a deterrent against using larger albeit realistic benchmarks [24], but running toy workloads on small disks is not sufficient. One obvious solution is to continually upgrade one's storage capacity. However, it is an expensive, and perhaps an infeasible solution to justify the costs and overheads solely for benchmarking.

Storage emulators such as Memulator [10] prove extremely useful for such scenarios – they let us prototype the "future" by pretending to plug in bigger, faster storage systems and run real workloads against them. Memulator, in fact, makes a strong case for storage emulation as *the* performance evaluation methodology of choice. But emulators are particularly tough: if they are to be big, they have to use existing storage (and thus are slow); if they are to be fast, they have to be run out of memory (and thus they are small).

The challenge we face is how can we get the best of both worlds? To address this problem, we have developed David, a "scale down" emulator that allows one to run large workloads by *scaling down* the storage requirements transparently to the workload. David makes it practical to experiment with benchmarks that were otherwise infeasible to run on a given system.

Our observation is that in many cases, the benchmark application does not care about the contents of individual files, but only about the structure and properties of the metadata that is being stored on disk. In particular, for the purposes of benchmarking, many applications do not write or read file contents at all (*e.g.*, `fsck`); the ones that do, often do not care what the contents are as long as *some* valid content is made available (*e.g.*, backup software). Since file data constitutes a significant fraction of the total file system size, ranging anywhere from 90 to 99% depending on the actual file-system image [3] avoiding the need to store file data has the potential to significantly reduce the required storage capacity during benchmarking.

The key idea in David is to create a "compressed" version of the original file-system image for the purposes of benchmarking. In the compressed image, unneeded user data blocks are omitted using novel classification techniques to distinguish data from metadata at scale; file system metadata blocks (e.g., inodes, directories and indirect blocks) are stored compactly on the available backing store. The primary benefit of the compressed image is to reduce the storage capacity required to run any given workload. To ensure that applications remain unaware of this interposition, whenever necessary, David synthetically generates file data on the fly; metadata I/O is redirected and accessed appropriately. David works under any file system; we demonstrate this using ext3 [25] and btrfs [26], two file systems very different in design.

Since David alters the original I/O patterns, it needs to model the runtime of the benchmark workload on the original uncompressed image. David uses an in-kernel model of the disk and storage stack to determine the run times of all individual requests as they would have executed on the uncompressed image. The model pays special attention to accurately modeling the I/O request queues; we find that modeling the request queues is crucial for overall accuracy, especially for applications issuing bursty I/O.

The primary mode of operation of David is the *timing-accurate* mode in which after modeling the runtime, an appropriate delay is inserted before returning to the application. A secondary *speedup* mode is also available in which the storage model returns instantaneously after computing the time taken to run the benchmark on the uncompressed disk; in this mode David offers the potential to reduce application runtime and speedup the benchmark itself. In this paper we discuss and evaluate David in the timing-accurate mode.

David allows one to run benchmark workloads that require file-system images orders of magnitude larger than the available backing store while still reporting the runtime as it would have taken on the original image. We demonstrate that David even enables emulation of faster and multi-disk systems like RAID using a small amount of memory. David can also aid in running large benchmarks on storage devices that are expensive or not even available in the market as it requires only a model of the non-existent storage device; for example, one can use a modified version of David to run benchmarks on a hypothetical 1TB SSD.

We believe David will be useful to file and storage developers, application developers, and users looking to benchmark these systems at scale. Developers often like to evaluate a prototype implementation at larger scales to identify performance bottlenecks, fine-tune optimizations, and make design decisions; analyses at scale often reveal interesting and critical insights into the system [16]. David can help obtain approximate performance estimates within limits of its modeling error. For example, how does one measure performance of a file system on a multi-disk multi-TB mirrored RAID configuration without having access to one? An end-user looking to select an application that works best at larger scale may also use David for emulation. For example, which anti-virus application scans a terabyte file system the fastest?

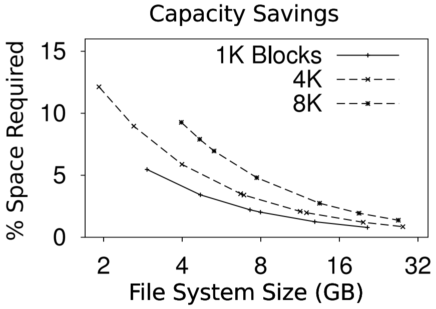


Figure 1: **Capacity Savings.** *Shows the savings in storage capacity if only metadata is stored, with varying file-size distribution modeled by (μ, σ) parameters of a lognormal distribution, (7.53, 2.48) and (8.33, 3.18) for the two extremes.*

One challenge in building David is how to deal with scale as we experiment with larger file systems containing many more files and directories. Figure 1 shows the percentage of storage space occupied by metadata alone as compared to the total size of the file-system image written; the different file-system images for this experiment were generated by varying the file size distribution using Impressions [2]. Using publicly available data on file-system metadata [4], we analyzed how file-size distribution changes with file systems of varying sizes.

We found that larger file systems not only had more files, they also had larger files. For this experiment, the parameters of the lognormal distribution controlling the file sizes were changed along the x-axis to generate progressively larger file systems with larger files therein. The relatively small fraction belonging to metadata (roughly 1 to 10%) as shown on the y-axis demonstrates the potential savings in storage capacity made possible if only metadata blocks are stored; David is designed to take advantage of this observation.

For workloads like PostMark, `mkfs`, `Filebench WebServer`, `Filebench VarMail`, and other microbenchmarks, we find that David delivers on its promise in reducing the required storage size while still accurately predicting the benchmark runtime for both ext3 and btrfs. The storage model within David is fairly accurate in spite of operating in real-time within the kernel, and for most workloads predicts a runtime within 5% of the actual runtime. For example, for the Filebench webserver workload, David provides a 1000-fold reduction in required storage capacity and predicts a runtime within 0.08% of the actual.

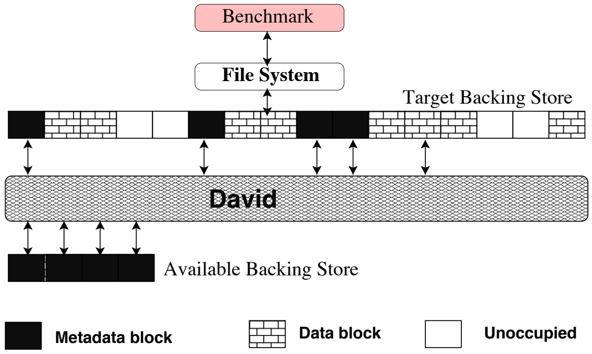


Figure 2: **Metadata Remapping and Data Squashing in David.** *The figure shows how metadata gets remapped and data blocks are squashed. The disk image above David is the* **target** *and the one below it is the* **available**.

2 David Overview

2.1 Design Goals for David

- **Scalability:** Emulating a large device requires David to maintain additional data structures and mimic several operations; our goal is to ensure that it works well as the underlying storage capacity scales.
- **Model accuracy:** An important goal is to model a storage device and accurately predict performance. The model should not only characterize the physical characteristics of the drive but also the interactions under different workload patterns.
- **Model overhead:** Equally important to being accurate is that the model imposes minimal overhead; since the model is inside the OS and runs concurrently with workload execution, it is required to be fairly fast.
- **Emulation flexibility:** David should be able to emulate different disks, storage subsystems, and multi-disk systems through appropriate use of backing stores.
- **Minimal application modification:** It should allow applications to run unmodified without knowing the significantly less capacity of the storage system underneath; modifications can be performed in limited cases only to improve ease of use but never as a necessity.

2.2 David Design

David exports a fake storage stack including a fake device of a much higher capacity than available. For the rest of the paper, we use the terms *target* to denote the hypothetical larger storage device, and *available* to denote the physically available system on which David is running, as shown in Figure 2. It also shows a schematic of how David makes use of metadata remapping and data squashing to free up a large percentage of the required storage space; a much smaller backing store can now service the requests of the benchmark.

David is implemented as a pseudo-device driver that is situated below the file system and above the backing store, interposing on all I/O requests. Since the driver appears as a regular device, a file system can be created and mounted on it. Being a loadable module, David can be used without any change to the application, file system or the kernel. Figure 3 presents the architecture of David with all the significant components and also shows the different types of requests that are handled within. We now describe the components of David.

First, the Block Classifier is responsible for classifying blocks addressed in a request as data or metadata and preventing I/O requests to data blocks from going to the backing store. David intercepts all writes to data blocks, records the block address if necessary, and discards the actual write using the Data Squasher. I/O requests to metadata blocks are passed on to the *Metadata Remapper*.

Second, the Metadata Remapper is responsible for laying out metadata blocks more efficiently on the backing store. It intercepts all write requests to metadata blocks, generates a remapping for the set of blocks addressed, and writes out the metadata blocks to the remapped locations. The remapping is stored in the Metadata Remapper to service subsequent reads.

Third, writes to data blocks are not saved, but reads to these blocks could still be issued by the application; in order to allow applications to run transparently, the Data Generator is responsible for generating synthetic content to service subsequent reads to data blocks that were written earlier and discarded. The Data Generator contains a number of built-in schemes to generate different kinds of content and also allows the application to provide hints to generate more tailored content (*e.g.*, binary files).

Finally, by performing the above-mentioned tasks David modifies the original I/O request stream. These modifications in the I/O traffic substantially change the application runtime rendering it useless for benchmarking. The Storage Model carefully models the (potentially different) target storage subsystem underneath to predict the benchmark runtime on the target system. By doing so in an online fashion with little overhead, the Storage Model makes it feasible to run large workloads in a space and time-efficient manner. The individual components are discussed in detail in §3 through §6.

2.3 Choice of Available Backing Store

David is largely agnostic to the choice of the backing store for available storage: HDDs, SSDs, or memory can be used depending on the performance and capacity requirements of the target device being emulated. Through a significant reduction in the number of device I/Os, David compensates for its internal book-keeping overhead and also for small mismatches between the emulated and available device. However, if one wishes to

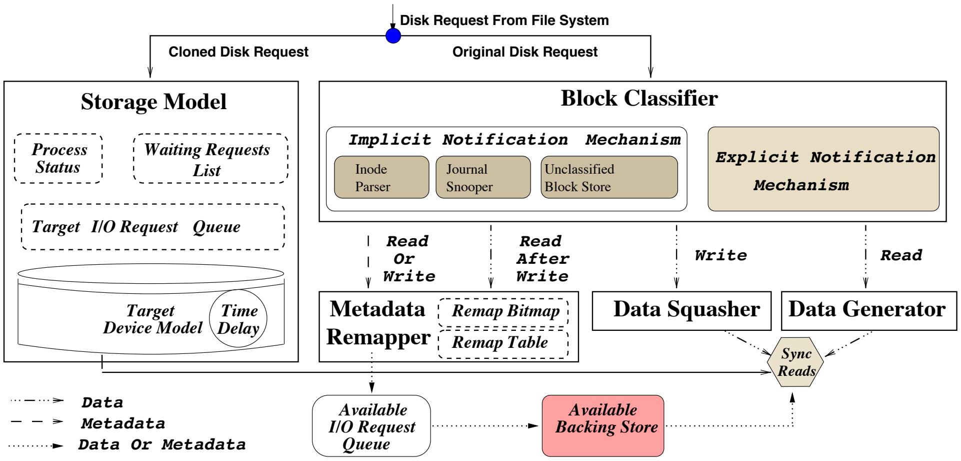


Figure 3: **David Architecture.** *Shows the components of David and the flow of requests handled within.*

emulate a device much faster than the available device, using memory is a safer option. For example, as shown in §6.3, David successfully emulates a RAID-1 configuration using a limited amount of memory. If the performance mismatch is not significant, a hard disk as backing store provides much greater scale in terms of storage capacity. Throughout the paper, "available storage" refers to the backing store in a generic sense.

3 Block Classification

The primary requirement for David to prevent data writes using the Data Squasher is the ability to classify a block as metadata or data. David provides both implicit and explicit block classification. The implicit approach is more laborious but provides a flexible approach to run unmodified applications and file systems. The explicit notification approach is straightforward and much simpler to implement, albeit at the cost of a small modification in the operating system or the application; both are available in David and can be chosen according to the requirements of the evaluator. The implicit approach is demonstrated using ext3 and the explicit approach using btrfs.

3.1 Implicit Type Detection

For ext2 and ext3, the majority of the blocks are statically assigned for a given file system size and configuration at the time of file system creation; the allocation for these blocks doesn't change during the lifetime of the file system. Blocks that fall in this category include the super block, group descriptors, inode and data bitmaps, inode blocks and blocks belonging to the journal; these blocks are relatively straightforward to classify based on their on-disk location, or their Logical Block Address (LBA). However, not all blocks are statically assigned; dynamically-allocated blocks include directory, indirect (single, double, or triple indirect) and data blocks. Unless all blocks contain some self-identification information, in order to accurately classify a dynamically allocated block, the system needs to track the inode pointing to the particular block to infer its current status.

Implicit classification is based on prior work on Semantically-Smart Disk Systems (SDS) [21]; an SDS employs three techniques to classify blocks: *direct* and *indirect* classification, and *association*. With direct classification, blocks are identified simply by their location on disk. With indirect classification, blocks are identified only with additional information; for example, to identify directory data or indirect blocks, the corresponding inode must also be examined. Finally, with association, a data block and its inode are connected.

There are two significant additional challenges David must address. First, as opposed to SDS, David has to ensure that no metadata blocks are ever misclassified. Second, benchmark scalability introduces additional memory pressure to handle delayed classification. In this paper we only discuss our new contributions (the original SDS paper provides details of the basic block-classification mechanisms).

3.1.1 Unclassified Block Store

To infer when a file or directory is allocated and deallocated, David tracks writes to inode blocks, inode bitmaps and data bitmaps; to enumerate the indirect and directory blocks that belong to a particular file or directory, it uses the contents of the inode. It is often the case that the blocks pointed to by an inode are written out before the corresponding inode block; if a classification attempt is made when a block is being written, an indirect or directory block will be misclassified as an ordinary data block. This transient error is unacceptable for David since it leads to the "metadata" block being discarded prematurely and could cause irreparable damage to the file system. For example, if a directory or indirect block is accidentally discarded, it could lead to file system corruption.

To rectify this problem, David temporarily buffers in memory writes to all blocks which are as yet unclassified, inside the *Unclassified Block Store* (UBS). These write requests remain in the UBS until a classification is made possible upon the write of the corresponding inode. When a corresponding inode does get written, blocks that are classified as metadata are passed on to the Metadata Remapper for remapping; they are then written out to persistent storage at the remapped location. Blocks classified as data are discarded at that time. All entries in the UBS corresponding to that inode are also removed.

The UBS is implemented as a list of block I/O (*bio*) request structures. An extra reference to the memory pages pointed to by these bio structures is held by David as long they remain in the UBS; this reference ensures that these pages are not mistakenly freed until the UBS is able to classify and persist them on disk, if needed. In order to reduce the inode parsing overhead otherwise imposed for each inode write, David maintains a list of recently written inode blocks that need to be processed and uses a separate kernel thread for parsing.

3.1.2 Journal Snooping

Storing unclassified blocks in the UBS can cause a strain on available memory in certain situations. In particular, when ext3 is mounted on top of David in ordered journaling mode, all the data blocks are written to disk at journal-commit time but the metadata blocks are written to disk only at the checkpoint time which occurs much less frequently. This results in a temporary yet precarious build up of data blocks in the UBS even though they are bound to be squashed as soon as the corresponding inode is written; this situation is especially true when large files (*e.g.*, 10s of GB) are written. In order to ensure the overall scalability of David, handling large files and the consequent explosion in memory consumption is critical. To achieve this without any modification to the ext3 filesystem, David performs Journal Snooping in the block device driver.

David snoops on the journal commit traffic for inodes and indirect blocks logged within a committed transaction; this enables block classification even prior to checkpoint. When a journal-descriptor block is written as part of a transaction, David records the blocks that are being logged within that particular transaction. In addition, all journal writes within that transaction are cached in memory until the transaction is committed. After that, the inodes and their corresponding direct and indirect blocks are processed to allow block classification; the identified data blocks are squashed from the UBS and the identified metadata blocks are remapped and stored persistently. The challenge in implementing Journal Snooping was to handle the continuous stream of unordered journal blocks and reconstruct the journal transaction.

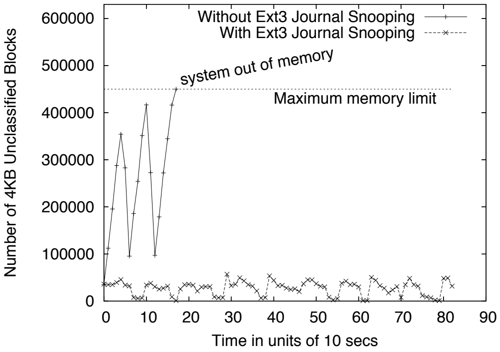


Figure 4: **Memory usage with Journal Snooping.**

Figure 4 compares the memory pressure with and without Journal Snooping demonstrating its effectiveness. It shows the number of 4 KB block I/O requests resident in the UBS sampled at 10 sec intervals during the creation of a 24 GB file on ext3; the file system is mounted on top of David in ordered journaling mode with a commit interval of 5 secs. This experiment was run on a dual core machine with 2 GB memory. Since this workload is data write intensive, without Journal Snooping, the system runs out of memory when around 450,000 bio requests are in the UBS (occupying roughly 1.8 GB of memory). Journal Snooping ensures that the memory consumed by outstanding bio requests does not go beyond a maximum of 240 MB.

3.2 Explicit Metadata Notification

David is meant to be useful for a wide variety of file systems; explicit metadata notification provides a mechanism to rapidly adopt a file system for use with David. Since data writes can come only from the benchmark application in user-space whereas metadata writes are issued by the file system, our approach is to identify the data blocks before they are even written to the file system. Our implementation of explicit notification is thus file-system agnostic – it relies on a small modification to the page cache to collect additional information. We demonstrate the benefits of this approach using btrfs, a file system quite unlike ext3 in design.

When an application writes to a file, David captures the pointers to the in-memory pages where the data content is stored, as it is being copied into the page cache. Subsequently, when the writes reach David, they are compared against the captured pointer addresses to decide whether the write is to metadata or data. Once the presence is tested, the pointer is removed from the list since the same page can be reused for metadata writes in the future.

There are certainly other ways to implement explicit notification. One way is to capture the checksum of the contents of the in-memory pages instead of the pointer to track data blocks. One can also modify the file system to explicitly flag the metadata blocks, instead of identifying data blocks with the page cache modification. We believe our approach is easier to implement, does not require any file system modification, and is also easier to extend to software RAID since parity blocks are automatically classified as metadata and not discarded.

4 Metadata Remapping

Since David exports a target pseudo device of much higher capacity to the file system than the available storage device, the bio requests issued to the pseudo device will have addresses in the full target range and thus need to be suitably remapped. For this purpose, David maintains a remap table called Metadata Remapper which maps "target" addresses to "available" addresses. The Metadata Remapper can contain an entry either for one metadata block (*e.g.*, super block), or a range of metadata blocks (*e.g.*, group descriptors); by allowing an arbitrary range of blocks to be remapped together, the Metadata Remapper provides an efficient translation service that also provides scalability. Range remapping in addition preserves sequentiality of the blocks if a disk is used as the backing store. In addition to the Metadata Remapper, a *remap bitmap* is maintained to keep track of free and used blocks on the available physical device; the remap bitmap supports allocation both of a single remapped block and a range of remapped blocks.

The destination (or remapped) location for a request is determined using a simple algorithm which takes as input the number of contiguous blocks that need to be remapped and finds the first available chunk of space from the *remap bitmap*. This can be done statically or at runtime; for the ext3 file system, since most of the blocks are statically allocated, the remapping for these blocks can also be done statically to improve performance. Subsequent writes to other metadata blocks are remapped dynamically; when metadata blocks are deallocated, corresponding entries from the Metadata Remapper and the remap bitmap are removed. From our experience, this simple algorithm lays out blocks on disk quite efficiently. More sophisticated allocation algorithms based on locality of reference can be implemented in the future.

5 Data Generator

David services the requirements of systems oblivious to file content with data squashing and metadata remapping. However, many real applications care about file content; the Data Generator with David is responsible for generating synthetic content to service read requests to data blocks that were previously discarded. Different systems can have different requirements for the file content and the Data Generator has various options to choose from; figure 5 shows some examples of the different types of content that can be generated.

Many systems that read back previously written data do not care about the *specific* content within the files as long as there is *some* content (*e.g.*, a file-system backup utility, or the Postmark benchmark). Much in the same way as failure-oblivious computing generates values to service reads to invalid memory while ignoring invalid writes [18], David randomly generates content to service out-of-bound read requests.

Some systems may expect file contents to have valid syntax or semantics; the performance of these systems depend on the actual content being read (*e.g.*, a desktop search engine for a file system, or a spell-checker). For such systems, naive content generation would either crash the application or give poor benchmarking results. David produces valid file content leveraging prior work on generating file-system images [2].

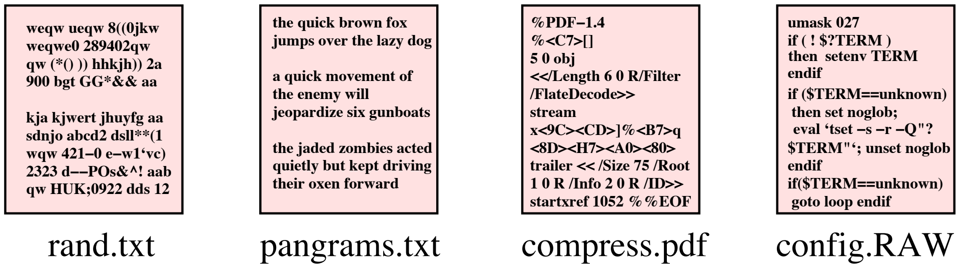
Finally, some systems may expect to read back data exactly as they wrote earlier (*i.e.*, a read-after-write or RAW dependency) or expect a precise structure that cannot be generated arbitrarily (*e.g.*, a binary file or a configuration file). David provides additional support to run these demanding applications using the *RAW Store*, designed as a cooperative resource visible to the user and configurable to suit the needs of different applications.

Our current implementation of RAW Store is very simple: in order to decide which data blocks need to be stored persistently, David requires the application to supply a list of the relevant file paths. David then looks up the inode number of the files and tracks all data blocks pointed to by these inodes, writing them out to disk using the Metadata Remapper just as any metadata block. In the future, we intend to support more nuanced ways to maintain the RAW Store; for example, specifying directories instead of files, or by using *Memoization* [14].

For applications that must exactly read back a significant fraction of what they write, the scalability advantage of David diminishes; in such cases the benefits are primarily from the ability to emulate new devices.

6 Storage Model and Emulation

Not having access to the target storage system requires David to precisely capture the behavior of the entire storage stack with all its dependencies through a model. The storage system modeled by David is the *target* system and the system on which it runs is the *available* system. David emulates the behavior of the target disk by sending requests to the available disk (for persistence) while simultaneously sending the *target request stream* to the Storage Model; the model computes the time that would have taken for the request to finish on the target system and introduces an appropriate delay in the actual request


rand.txt
pangrams.txt
compress.pdf
config.RAW

```
weqw ueqw 8((0jkw
weqwe0 289402qw
qw (*() )) hhkjh)) 2a
900 bgt GG*&& aa

kja kjwert jhuyfg aa
sdnjo abcd2 dsll**(1
wqw 421–0 e–w1‘vc)
2323 d––POs&^! aab
qw HUK;0922 dds 12
```

```
the quick brown fox
jumps over the lazy dog

a quick movement of
the enemy will
jeopardize six gunboats

the jaded zombies acted
quietly but kept driving
their oxen forward
```

```
%PDF–1.4
%<C7>[]
5 0 obj
<</Length 6 0 R/Filter
/FlateDecode>>
stream
x<9C><CD>]%<B7>q
<8D><H7><A0><80>
trailer << /Size 75 /Root
1 0 R /Info 2 0 R /ID>>
startxref 1052 %%EOF
```

```
umask 027
if ( ! $?TERM )
then  setenv TERM
endif
if ($TERM==unknown)
 then set noglob;
 eval ‘tset –s –r –Q"?
$TERM"‘; unset noglob
endif
if($TERM==unknown)
 goto loop endif
```

Figure 5: **Examples of content generation by Data Generator.** *The figure shows a randomly generated text file, a text file with semantically meaningful content, a well-formatted PDF file, and a config file with precise syntax to be stored in the RAW Store.*

Parameter	H1	H2	Parameter	H1	H2
Disk size	80 GB	1 TB	Cache segments	11	500
Rotational Speed	7200 RPM	7200 RPM	Cache R/W partition	Varies	Varies
Number of cylinders	88283	147583	Bus Transfer	133 MBps	133 MBps
Number of zones	30	30	Seek profile(long)	$3800+(cyl*116)/10^3$	$3300+(cyl*5)/10^6$
Sectors per track	567 to 1170	840 to 1680	Seek profile(short)	$300+\sqrt{(cyl*2235)}$	$700+\sqrt{cyl}$
Cylinders per zone	1444 to 1521	1279 to 8320	Head switch	1.4 ms	1.4 ms
On-disk cache size	2870 KB	300 MB	Cylinder switch	1.6 ms	1.6 ms
Disk cache segment	260 KB	600 KB	Dev driver req queue*	128-160	128-160
Req scheduling*	FIFO	FIFO	Req queue timeout*	3 ms (unplug)	3 ms (unplug)

Table 1: **Storage Model Parameters in David.** *Lists important parameters obtained to model disks Hitachi HDS728080PLA380 (H1) and Hitachi HDS721010KLA330 (H2). *denotes parameters of I/O request queue (IORQ).*

stream before returning control. Figure 3 presented in §2 shows this setup more clearly.

As a general design principle, to support low-overhead modeling without compromising accuracy, we avoid using any technique that either relies on storing empirical data to compute statistics or requires table-based approaches to predict performance [6]; the overheads for such methods are directly proportional to the amount of runtime statistics being maintained which in turn depends on the size of the disk. Instead, wherever applicable, we have adopted and developed analytical approximations that did not slow the system down; our resulting models are sufficiently lean while being fairly accurate.

To ensure portability of our models, we have refrained from making device-specific optimizations to improve accuracy; we believe current models in David are fairly accurate. The models are also adaptive enough to be easily configured for changes in disk drives and other parameters of the storage stack. We next present some details of the disk model and the storage stack model.

6.1 Disk Model

David's disk model is based on the classical model proposed by Ruemmler and Wilkes [19], henceforth referred as the RW model. The disk model contains information about the disk geometry (*i.e.*, platters, cylinders and zones) and maintains the current disk head position; using these sources it models the disk seek, rotation, and transfer times for any request. The disk model also keeps track of the effects of disk caches (track prefetching, write-through and write-back caches). In the future, it will be interesting to explore using Disksim for the disk model. Disksim is a detailed user-space disk simulator which allows for greater flexibility in the types of device properties that can be simulated along with their degree of detail; we will need to ensure it does not appreciably slow down the emulation when used without memory as backing store.

6.1.1 Disk Drive Profile

The disk model requires a number of drive-specific parameters as input, a list of which is presented in the first column of Table 1; currently David contains models for two disks: the Hitachi HDS728080PLA380 80 GB disk, and the Hitachi HDS721010KLA330 1 TB disk. We have verified the parameter values for both these disks through carefully controlled experiments. David is envisioned for use in environments where the target drive itself may not be available; if users need to model additional drives, they need to supply the relevant parameters. Disk seeks, rotation time and transfer times are modeled much in the same way as proposed in the RW model. The actual parameter values defining the above properties are specific to a drive; empirically obtained values for the two disks we model are shown in Table 1.

6.1.2 Disk Cache Modeling

The drive cache is usually small (few hundred KB to a few MB) and serves to cache reads from the disk media to service future reads, or to buffer writes. Unfortunately, the drive cache is one of the least specified components as well; the cache management logic is low-level firmware code which is not easy to model.

David models the number and size of segments in the disk cache and the number of disk sector-sized slots in each segment. Partitioning of the cache segments into read and write caches, if any, is also part of the information contained in the disk model. David models the read cache with a FIFO eviction policy. To model the effects of write caching, the disk model maintains statistics on the current size of writes pending in the disk cache and the time needed to flush these writes out to the media. Write buffering is simulated by periodically emptying a fraction of the contents of the write cache during idle periods of the disk in between successive foreground requests. The cache is modeled with a write-through policy and is partitioned into a sufficiently large read cache to match the read-ahead behavior of the disk drive.

6.2 Storage Stack Model

David also models the I/O request queues (IORQs) maintained in the OS; Table 1 lists a few of its important parameters. While developing the Storage Model, we found that accurately modeling the behavior of the IORQs is crucial to predict the target execution time correctly. The IORQs usually have a limit on the maximum number of requests that can be held at any point; processes that try to issue an I/O request when the IORQ is full are made to wait. Such waiting processes are woken up when an I/O issued to the disk drive completes, thereby creating an empty slot in the IORQ. Once woken up, the process is also granted privilege to batch a constant number of additional I/O requests even when the IORQ is full, as long as the total number of requests is within a specified upper limit. Therefore, for applications issuing bursty I/O, the time spent by a request in the IORQ can outweigh the time spent at the disk by several orders of magnitude; modeling the IORQs is thus crucial for overall accuracy.

Disk requests arriving at David are first enqueued into a *replica queue* maintained inside the Storage Model. While being enqueued, the disk request is also checked for a possible *merge* with other pending requests: a common optimization that reduces the number of total requests issued to the device. There is a limit on the number of disk requests that can be merged into a single disk request; eventually merged disk requests are dequeued from the *replica queue* and dispatched to the disk model to obtain the service time spent at the drive. The *replica queue* uses the same request scheduling policy as the target IORQ.

6.3 RAID Emulation

David can also provide effective RAID emulation. To demonstrate simple RAID configurations with David, each component disk is emulated using a memory-backed "compressed" device underneath software RAID. David exports multiple block devices with separate major and minor numbers; it differentiates requests to different devices using the major number. For the purpose of performance benchmarking, David uses a single memory-based backing store for all the compressed RAID devices. Using multiple threads, the Storage Model maintains separate state for each of the devices being emulated. Requests are placed in a single request queue tagged with a device identifier; individual Storage Model threads for each device fetch one request at a time from this request queue based on the device identifier. Similar to the single device case, the servicing thread calculates the time at which a request to the device should finish and notifies completion using a callback.

David currently only provides mechanisms for simple software RAID emulation that do not need a model of a software RAID itself. New techniques might be needed to emulate more complex commercial RAID configurations, for example, commercial RAID settings using a hardware RAID card.

7 Evaluation

We seek to answer four important questions. First, what is the accuracy of the Storage Model? Second, how accurately does David predict benchmark runtime and what storage space savings does it provide? Third, can David scale to large target devices including RAID? Finally, what is the memory and CPU overhead of David?

7.1 Experimental Platform

We have developed David for the Linux operating system. The hard disks currently modeled are the 1 TB Hitachi HDS721010KLA330 (referred to as D_{1TB}) and the 80 GB Hitachi HDS728080PLA380 (referred to as D_{80GB}); table 1 lists their relevant parameters. Unless specified otherwise, the following hold for all the experiments: (1) machine used has a quad-core Intel processor and 4GB RAM running Linux 2.6.23.1 (2) ext3 file system is mounted in ordered-journaling mode with a commit interval of 5 sec (3) microbenchmarks were run directly on the disk without a file system (4) David predicts the benchmark runtime for a target D_{1TB} while in fact running on the available D_{80GB} (5) to validate accuracy, David was instead run directly on D_{1TB}.

7.2 Storage Model Accuracy

First, we validate the accuracy of Storage Model in predicting the benchmark runtime on the target system. Since our aim is to validate the accuracy of the Storage Model alone, we run David in a *model only* mode where we disable block classification, remapping and data squashing. David just passes down the requests that it receives to the available request queue below. We run

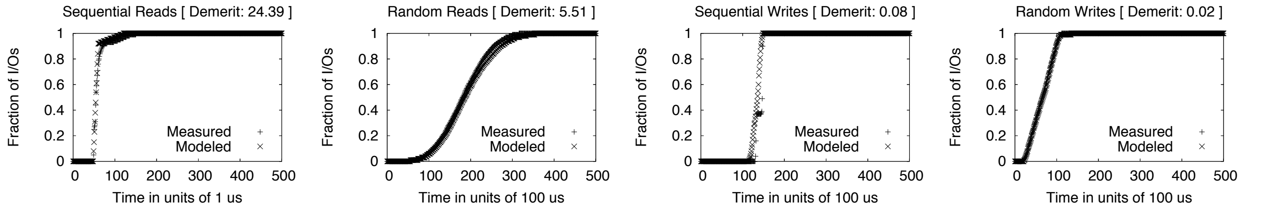


Figure 6: **Storage Model accuracy for Sequential and Random Reads and Writes.** *The graph shows the cumulative distribution of measured and modeled times for sequential and random reads and writes.*

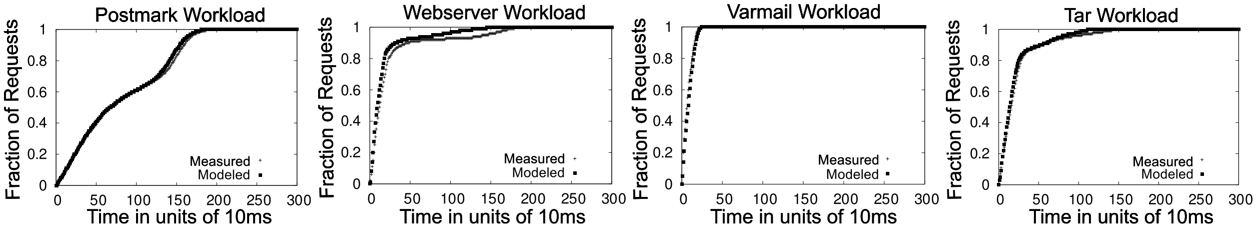


Figure 7: **Storage Model accuracy.** *The graphs show the cumulative distribution of measured and modeled times for the following workloads from left to right: Postmark, Webserver, Varmail and Tar.*

	Implicit Classification – Ext3						Explicit Notification – Btrfs		
Benchmark Workload	Original Storage (KB)	David Storage (KB)	Storage Savings (%)	Original Runtime (Secs)	David Runtime (Secs)	Runtime Error (%)	Original Runtime (Secs)	David Runtime (Secs)	Runtime Error (%)
mkfs	976762584	7900712	99.19	278.66	281.81	1.13	-	-	-
imp	11224140	18368	99.84	344.18	339.42	-1.38	327.294	324.057	0.99
tar	21144	628	97.03	257.66	255.33	-0.9	146.472	135.014	7.8
grep	-	-	-	250.52	254.40	1.55	141.960	138.455	2.47
virus scan	-	-	-	55.60	47.95	-13.75	27.420	31.555	15.08
find	-	-	-	26.21	26.60	1.5	-	-	-
du	-	-	-	102.69	101.36	-1.29	-	-	-
postmark	204572	404	99.80	33.23	29.34	-11.69	22.709	22.243	2.05
webserver	3854828	3920	99.89	127.04	126.94	-0.08	125.611	126.504	0.71
varmail	7852	3920	50.07	126.66	126.27	-0.31	126.019	126.478	0.36
sr	-	-	-	40.32	44.90	11.34	40.32	44.90	11.34
rr	-	-	-	913.10	935.46	2.45	913.10	935.46	2.45
sw	-	-	-	57.28	58.96	2.93	57.28	58.96	2.93
rw	-	-	-	308.74	291.40	-5.62	308.74	291.40	-5.62

Table 2: **David Performance and Accuracy.** *Shows savings in capacity, accuracy of runtime prediction, and the overhead of storage modeling for different workloads. Webserver and varmail are generated using FileBench; virus scan using AVG.*

David on top of D_{1TB} and set the target drive to be the same. Note that the available system is the *same* as the target system for these experiments since we only want to compare the measured and modeled times to validate the accuracy of the Storage Model. Each block request is traced along its path from David to the disk drive and back. This is done in order to measure the total time that the request spends in the available IORQ and the time spent getting serviced at the available disk. These measured times are then compared with the modeled times obtained from the Storage Model.

Figure 6 shows the Storage Model accuracy for four micro-workloads: sequential and random reads, and sequential and random writes; these micro-workloads have demerit figures of 24.39, 5.51, 0.08, and 0.02 respectively, as computed using the Ruemmler and Wilkes methodology [19]. The large demerit for sequential reads is due to a variance in the available disk's cache-read times; modeling the disk cache in greater detail in the future could potentially avoid this situation. However, sequential read requests do not contribute to a measurably large error in the total modeled runtime; they often hit the disk cache and have service times less than 500 microseconds while other types of disk requests take around 20 to 35 milliseconds to get serviced. Any inaccuracy in the modeled times for sequential reads is negligible when compared to the service times of other types of disk requests; we thus chose to not make the disk-cache model more complex for the sake of sequential reads.

Figure 7 shows the accuracy for four different macro workloads and application kernels: Postmark [13], webserver (generated using FileBench [15]), Varmail (mail server workload using FileBench), and a Tar workload (copy and untar of the linux kernel of size 46 MB).

The FileBench Varmail workload emulates an NFS mail server, similar to Postmark, but is multi-threaded instead. The Varmail workload consists of a set of open/read/close, open/append/close and deletes in a single directory, in a multi-threaded fashion. The FileBench webserver workload comprises of a mix of open/read/close of multiple files in a directory tree. In addition, to simulate a webserver log, a file append operation is also issued. The workload consists of 100 threads issuing 16 KB appends to the weblog every 10 reads.

Overall, we find that storage modeling inside David is quite accurate for all workloads used in our evaluation. The total modeled time as well as the distribution of the individual request times are close to the total measured time and the distribution of the measured request times.

7.3 David Accuracy

Next, we want to measure how accurately David predicts the benchmark runtime. Table 2 lists the accuracy and storage space savings provided by David for a variety of benchmark applications for both ext3 and btrfs. We have chosen a set of benchmarks that are commonly used and also stress various paths that disk requests take within David. The first and second columns of the table show the storage space consumed by the benchmark workload without and with David. The third column shows the percentage savings in storage space achieved by using David. The fourth column shows the original benchmark runtime without David on D_{1TB}. The fifth column shows the benchmark runtime with David on D_{80GB}. The sixth column shows the percentage error in the prediction of the benchmark runtime by David. The final three columns show the original and modeled runtime, and the percentage error for the btrfs experiments; the storage space savings are roughly the same as for ext3. The *sr*, *rr*, *sw*, and *rw* workloads are run directly on the raw device and hence are independent of the file system.

mkfs creates a file system with a 4 KB block size over the 1 TB target device exported by David. This workload only writes metadata and David remaps writes issued by *mkfs* sequentially starting from the beginning of D_{80GB}; no data squashing occurs in this experiment.

imp creates a realistic file-system image of size 10 GB using the publicly available Impressions tool [2]. A total of 5000 regular files and 1000 directories are created with an average of 10.2 files per directory. This workload is a data-write intensive workload and most of the issued writes end up being squashed by David.

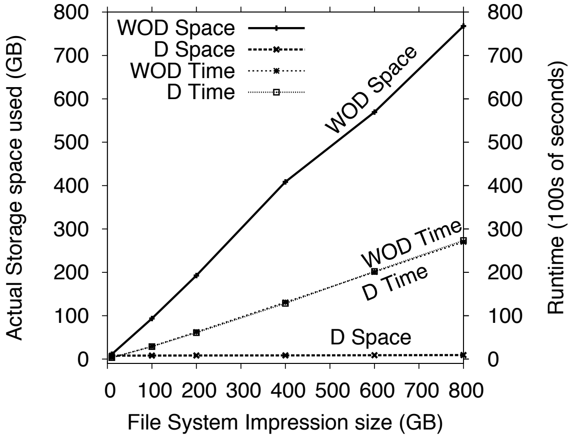


Figure 8: **Storage Space Savings and Model Accuracy.** *The "Space" lines show the savings in storage space achieved when using David for the* impressions *workload with file-system images of varying sizes until 800GB; "Time" lines show the accuracy of runtime prediction for the same workload.* **WOD***: space/time without David,* **D***: space/time with David.*

tar uses the GNU tar utility to create a gzipped archive of the file-system image of size 10 GB created by *imp*; it writes the newly created archive in the same file system. This workload is a data read and data write intensive workload. The data reads are satisfied by the Data Generator without accessing the available disk, while the data writes end up being squashed.

grep uses the GNU grep utility to search for the expression "nothing" in the content generated by both *imp* and *tar*. This workload issues significant amounts of data reads and small amounts of metadata reads. *virus scan* runs the AVG virus scanner on the file-system image created by *imp*. *find* and *du* run the GNU find and GNU du utilities over the content generated by both *imp* and *tar*. These two workloads are metadata read only workloads.

David works well under both the implicit and explicit approaches demonstrating its usefulness across file systems. Table 2 shows how David provides tremendous savings in the required storage capacity, upwards of 99% (a 100-fold or more reduction) for most workloads. David also predicts benchmark runtime quite accurately. Prediction error for most workloads is less than 3%, although for a few it is just over 10%. The errors in the predicted runtimes stem from the relative simplicity of our in-kernel Disk Model; for example, it does not capture the layout of physical blocks on the magnetic media accurately. This information is not published by the disk manufacturers and experimental inference is not possible for ATA disks that do not have a command similar to the SCSI `mode page`.

7.4 David Scalability

David is aimed at providing scalable emulation using commodity hardware; it is important that accuracy is

Num Disks	Rand R	Rand W	Seq R	Seq W
Measured				
3	232.77	72.37	119.29	119.98
2	156.76	72.02	119.11	119.33
1	78.66	71.88	118.65	118.71
Modeled				
3	238.79	73.77	119.44	119.40
2	159.36	72.21	119.16	119.21
1	79.56	72.15	118.95	118.83

Table 3: **David Software RAID-1 Emulation.** *Shows IOPS for a software RAID-1 setup using David with memory as backing store; workload issues 20000 read and write requests through concurrent processes which equal the number of disks in the experiment. 1 disk experiments run w/o RAID-1.*

not compromised at larger scale. Figure 8 shows the accuracy and storage space savings provided by David while creating file-system images of 100s of GB. Using an available capacity of only 10 GB, David can model the runtime of Impressions in creating a realistic file-system image of 800 GB; in contrast to the linear scaling of the target capacity demanded, David barely requires any extra available capacity. David also predicts the benchmark runtime within a maximum of 2.5% error even with the huge disparity between target and available disks at the 800 GB mark, as shown in Figure 8.

The reason we limit these experiments to a target capacity of less than 1 TB is because we had access to only a terabyte sized disk against which we could validate the accuracy of David. Extrapolating from this experience, we believe David will enable one to emulate disks of 10s or 100s of TB given the 1 TB disk.

7.5 David for RAID

We present a brief evaluation and validation of software RAID-1 configurations using David. Table 3 shows a simple experiment where David emulates a multi-disk software RAID-1 (mirrored) configuration; each device is emulated using a memory-disk as backing store. However, since the multiple disks contain copies of the same block, a single physical copy is stored, further reducing the memory footprint. In each disk setup, a set of threads which equal in number to the number of disks issue a total of 20000 requests. David is able to accurately emulate the software RAID-1 setup upto 3 disks; more complex RAID schemes are left as part of future work.

7.6 David Overhead

David is designed to be used for benchmarking and not as a production system, thus scalability and accuracy are the more relevant metrics of evaluation; we do however want to measure the memory and CPU overhead of using David on the available system to ensure it is practical to use. All memory usage within David is tracked using several counters; David provides support to measure the memory usage of its different components using `ioctls`. To measure the CPU overhead of the Storage Model alone, David is run in the *model-only* mode where block classification, remapping and data squashing are turned off.

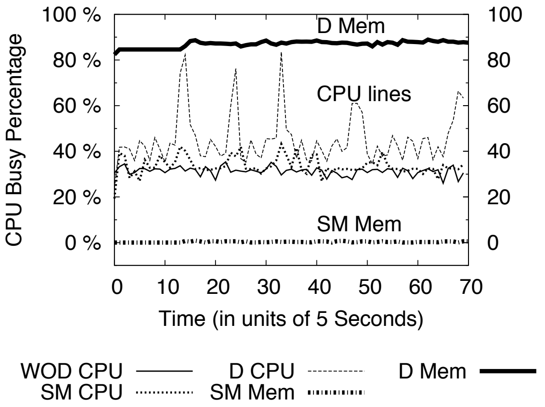


Figure 9: **David CPU and Memory Overhead.** *Shows the memory and percentage CPU consumption by David while creating a 10 GB file-system image using* impressions. **WOD CPU***: CPU without David,* **SM CPU***: CPU with Storage Model alone,* **D CPU***: total CPU with David,* **SM Mem***: Storage Model memory alone,* **D Mem***: total memory with David.*

In our experience with running different workloads, we found that the memory and CPU usage of David is acceptable for the purposes of benchmarking. As an example, Figure 9 shows the CPU and memory consumption by David captured at 5 second intervals while creating a 10 GB file-system image using Impressions. For this experiment, the Storage Model consumes less than 1 MB of memory; the average memory consumed in total by David is less than 90 MB, of which the pre-allocated cache used by the Journal Snooping to temporarily store the journal writes itself contributes 80 MB. Amount of CPU used by the Storage Model alone is insignificant, however implicit classification by the Block Classifier is the primary consumer of CPU using 10% on average with occasional spikes. The CPU overhead is not an issue at all if one uses explicit notification.

8 Related Work

Memulator [10] makes a great case for why storage emulation provides the unique ability to explore nonexistent storage components and take end-to-end measurements. Memulator is a “timing-accurate” storage emulator that allows a simulated storage component to be plugged into a real system running real applications. Memulator can use the memory of either a networked machine or the local machine as the storage media of the emulated disk, enabling full system evaluation of hypothetical storage devices. Although this provides flexibility in device emulation, high-capacity devices requires an equivalent amount of memory; David provides the necessary scalability to emulate such devices. In turn, David can benefit from the networked-emulation capabilities of Memulator in scenarios when either the host machine has limited CPU and memory resources, or when the interference of running David on the same machine competing for the same resources is unacceptable.

One alternate to emulation is to simply buy a larger capacity or newer device and use it to run the benchmarks. This is sometimes feasible, but often not desirable. Even if one buys a larger disk, in the future they would need an even larger one; David allows one to keep up with this arms race without always investing in new devices. Note that we chose 1 TB as the upper limit for evaluation in this paper because we could validate our results for that size. Having a large disk will also not address the issue of emulating much faster devices such as SSDs or RAID configurations. David emulates faster devices through an efficient use of memory as backing store.

Another alternate is to simulate the storage component under test; disk simulators like Disksim [7] allow such an evaluation flexibly. However, simulation results are often far from perfect [9] – they fail to capture system dependencies and require the generation of representative I/O traces which is a challenge in itself.

Finally, one might use analytical modeling for the storage devices; while very useful in some circumstances, it is not without its own set of challenges and limitations [20]. In particular, it is extremely hard to capture the interactions and complexities in real systems. Wherever possible, David does leverage well-tested analytical models for individual components to aid the emulation.

Both simulation and analytical modeling are complementary to emulation, perfectly useful in their own right. Emulation does however provide a reasonable middle ground in terms of flexibility and realism.

Evaluation of how well an I/O system scales has been of interest in prior research and is becoming increasingly more relevant [28]. Chen and Patterson proposed a "self-scaling" benchmark that scales with the I/O system being evaluated, to stress the system in meaningful ways [8]. Although useful for disk and I/O systems, the self-scaling benchmarks are not directly applicable for file systems. The evaluation of the XFS file system from Silicon Graphics uses a number of benchmarks specifically intended to test its scalability [23]; such an evaluation can benefit from David to employ even larger benchmarks with greater ease; SpecSFS [27] also contains some techniques for scalable workload generation.

Similar to our emulation of scale in a storage system, Gupta *et al*. from UCSD propose a technique called *time dilation* for emulating network speeds orders of magnitude faster than available [11]. Time dilation allows one to experiment with unmodified applications running on commodity operating systems by subjecting them to much faster network speeds than actually available.

A key challenge in David is the ability to identify data and meta-data blocks. Besides SDS [21], XN, the stable storage system for the Xok exokernel [12] dealt with similar issues. XN employed a template of metadata translation functions called *UDFs* specific to each file type. The responsibility of providing UDFs rested with the file system developer, allowing the kernel to handle arbitrary metadata layouts without understanding the layout itself. Specifying an encoding of the on-disk scheme can be tricky for a file system such as ReiserFS that uses dynamic allocation; however, in the future, David's metadata classification scheme can benefit from a more formally specified on-disk layout per file-system.

9 Conclusion

David is born out of the frustration in doing large-scale experimentation on realistic storage hardware – a problem many in the storage community face. David makes it practical to experiment with benchmarks that were otherwise infeasible to run on a given system, by transparently scaling down the storage capacity required to run the workload. The available backing store under David can be orders of magnitude smaller than the target device. David ensures accuracy of benchmarking results by using a detailed storage model to predict the runtime. In the future, we plan to extend David to include support for a number of other useful storage devices and configurations. In particular, the Storage Model can be extended to support flash-based SSDs using an existing simulation model [5]. We believe David will be a useful emulator for file and storage system evaluation.

10 Acknowledgments

We thank the anonymous reviewers and Rob Ross (our shepherd) for their feedback and comments, which have substantially improved the content and presentation of this paper. The first author thanks the members of the Storage Systems Group at NEC Labs for their comments and feedback.

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] GraySort Benchmark. http://sortbenchmark.org/FAQ.htm#gray.

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

[3] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A Five-Year Study of File-System Metadata. In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST '07)*, San Jose, California, February 2007.

[4] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A five-year study of file-system metadata: Microsoft longitudinal dataset. http://iotta.snia.org/traces/list/Static, 2007.

[5] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In *Proceedings of the Usenix Annual Technical Conference (USENIX '08)*, Boston, MA, June 2008.

[6] E. Anderson. Simple table-based modeling of storage devices. Technical Report HPL-SSP-2001-04, HP Laboratories, July 2001.

[7] J. S. Bucy and G. R. Ganger. The DiskSim Simulation Environment Version 3.0 Reference Manual. Technical Report CMU-CS-03-102, Carnegie Mellon University, January 2003.

[8] P. M. Chen and D. A. Patterson. A New Approach to I/O Performance Evaluation–Self-Scaling I/O Benchmarks, Predicted I/O Performance. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '93)*, pages 1–12, Santa Clara, California, May 1993.

[9] G. R. Ganger and Y. N. Patt. Using system-level models to evaluate i/o subsystem designs. *IEEE Trans. Comput.*, 47(6):667–678, 1998.

[10] J. L. Griffin, J. Schindler, S. W. Schlosser, J. S. Bucy, and G. R. Ganger. Timing-accurate Storage Emulation. In *Proceedings of the 1st USENIX Symposium on File and Storage Technologies (FAST '02)*, Monterey, California, January 2002.

[11] D. Gupta, K. Yocum, M. McNett, A. C. Snoeren, A. Vahdat, and G. M. Voelker. To infinity and beyond: time-warped network emulation. In *Proceedings of the 3rd conference on Networked Systems Design and Implementation (NSDI'06)*, San Jose, CA, 2006.

[12] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 52–65, Saint-Malo, France, October 1997.

[13] J. Katcher. PostMark: A New File System Benchmark. Technical Report TR-3022, Network Appliance Inc., oct 1997.

[14] J. Mayfield, T. Finin, and M. Hall. Using automatic memoization as a software engineering tool in real-world ai systems. *Artificial Intelligence for Applications, Conference on*, 0:87, 1995.

[15] R. McDougall. Filebench: Application level file system benchmark. http://www.solarisinternals.com/si/tools/filebench/index.php.

[16] E. L. Miller. Towards scalable benchmarks for mass storage systems. In *5th NASA Goddard Conference on Mass Storage Systems and Technologies*, 1996.

[17] E. Riedel, M. Kallahalla, and R. Swaminathan. A Framework for Evaluating Storage System Security. In *Proceedings of the 1st USENIX Symposium on File and Storage Technologies (FAST '02)*, pages 14–29, Monterey, California, January 2002.

[18] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and J. William S. Beebe. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francicso, CA, December 2004.

[19] C. Ruemmler and J. Wilkes. An Introduction to Disk Drive Modeling. *IEEE Computer*, 27(3):17–28, March 1994.

[20] E. Shriver. *Performance modeling for realistic storage devices.* PhD thesis, New York, NY, USA, 1997.

[21] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In

Proceedings of the 2nd USENIX Symposium on File and Storage Technologies (FAST '03), pages 73–88, San Francisco, California, April 2003.

[22] Standard Performance Evaluation Corporation. SPECmail2009 Benchmark. `http://www.spec.org/mail2009/`.

[23] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '96)*, San Diego, California, January 1996.

[24] A. Traeger and E. Zadok. How to cheat at benchmarking. In *USENIX FAST Birds of a feather session*, San Francisco, CA, February 2009.

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

[26] Wikipedia. Btrfs. en.wikipedia.org/wiki/Btrfs, 2009.

[27] M. Wittle and B. E. Keith. LADDIS: The next generation in NFS file server benchmarking. In *USENIX Summer*, pages 111–128, 1993.

[28] E. Zadok. File and storage systems benchmarking workshop. UC Santa Cruz, CA, May 2008.

Just-In-Time Analytics on Large File Systems

H. Howie Huang[1], Nan Zhang[1], Wei Wang[1], Gautam Das[2], and Alexander S. Szalay[3]

[1]George Washington University
[2]University of Texas at Arlington
[3]Johns Hopkins University

Abstract

As file systems reach the petabytes scale, users and administrators are increasingly interested in acquiring high-level analytical information for file management and analysis. Two particularly important tasks are the processing of aggregate and top-k queries which, unfortunately, cannot be quickly answered by hierarchical file systems such as ext3 and NTFS. Existing pre-processing based solutions, e.g., file system crawling and index building, consume a significant amount of time and space (for generating and maintaining the indexes) which in many cases cannot be justified by the infrequent usage of such solutions. In this paper, we advocate that user interests can often be sufficiently satisfied by *approximate* - i.e., statistically accurate - answers. We develop *Glance*, a just-in-time sampling-based system which, after consuming a small number of disk accesses, is capable of producing extremely accurate answers for a broad class of aggregate and top-k queries over a file system without the requirement of any prior knowledge. We use a number of real-world file systems to demonstrate the efficiency, accuracy and scalability of Glance.

1 Introduction

Today a file system with billions of files, millions of directories and petabytes of storage is no longer an exception [29]. As file systems grow, users and administrators are increasingly keen to perform complex queries [37, 47], such as “How many files have been updated since ten days ago?”, and “Which are the top five largest files that belong to John?”. The first is an example of *aggregate queries* which provide a high-level summary of all or part of the file system, while the second is *top-k queries* which locate the k files and/or directories that have the highest score according to a scoring function. Fast processing of aggregate and top-k queries are often needed by applications that require *just-in-time analytics* over large file systems, such as data management, archiving, enterprise surveillance, etc. The just-in-time requirement is defined by two properties: (1) file-system analytics must be completed within a short amount of time, and (2) the analyzer holds no prior knowledge (e.g., pre-processing results) of the file system being analyzed. For example, in order for a librarian to determine how to build an image archive from an external storage media (e.g., a Blue-ray disc), he/she may have to first estimate the total size of picture files stored on the external media - the librarian needs to complete data analytics quickly, over an alien file system that has never been seen before.

Unfortunately, hierarchical file systems (e.g., ext3 and NTFS) are not well equipped for the task of just-in-time analytics [43]. The deficiency is in general due to the lack of a *global view* (i.e., high-level statistics) of metadata information (e.g., size, creation, access and modification time). For efficiency concerns, a hierarchical file system is usually designed to limit the update of metadata information to individual files and/or the immediately preceding directories, leading to localized views. For example, while the last modification time of an individual file is easily retrievable, the last modification time of files that belong to user John is difficult to obtain because such metadata information is not available at the global level.

Currently, there are two approaches for generating high-level statistics from a hierarchical file system, and thereby answering aggregate and top-k queries: (1) scanning the file system upon the arrival of each query, e.g., the *find* command in Linux, which is inefficient for large file systems. While storage capacity increases ∼60% per year, storage throughput and latency have much slower improvements, thus the amount of time required to scan an off-the-shelf hard drive or external storage media has increased significantly over time to become infeasible for just-in-time analytics. The above-mentioned image-archiving application is a typical example, as it is usually impossible to completely scan an alien Blue-ray disc within a short amount of time. (2) utilizing pre-built indexes which are regularly updated [3, 7, 26, 32, 36, 40]. Many desktop search products, e.g., Google Desktop [23] and Beagle [5], belong to this category. While this approach is capable of fast query processing once the (slow) index building process is complete, it may not be suitable or applicable to many just-in-time applications:

- Index building can be unrealistic for many applications that require just-in-time analytics over an alien file system. An example is enterprise surveillance [35], where portable machines and storage devices must be quickly examined before being allowed to join the enterprise network.
- Even if index can be built up-front, its significant cost may not be justifiable if the index is not frequently used afterwards. Unfortunately, this is common for some large file systems, e.g., storage archives or scratch data for scientific applications rarely require the global search function offered by the index, and may only need analytical queries to be answered infrequently (e.g., once every few days). In this case, building and updating an index is often an overkill given the high amortized cost.
- There are also other limitations of maintaining an index. For example, prior work [46] has shown that even after a file has been completely removed (from both the file system and the index), the (former) existence of this file can still be inferred from the index structure. Thus, a file system owner may choose to avoid building an index for privacy concerns.

To enable just-in-time analytics, one must be able to perform an on-the-fly processing of analytical queries, over traditional file systems that normally have insufficient metadata to support such complex queries. We achieve this goal by striking a balance between query answer accuracy and cost - providing approximate (i.e., statistically accurate) answers which, with a high confidence level, reside within a close distance from the precise answer. For example, when a user wants to count the number of files in a directory (and all of its subdirectories), an approximate answer of $105,000$ or $95,000$, compared with the real answer of $100,000$, makes little difference to the high-level knowledge desired by the user. In general, the higher cost a user is willing to pay for answering a query, more accurate the answer can be.

To this end, we design and develop *Glance*, a just-in-time query processing system which produces accurate query answers based on a small number of samples (files or folders) that can be collected from a very large file system with a few disk accesses. Glance is file-system agnostic, i.e., it can be applied instantly over any new file system and work seamlessly with the tree structure of the system. Glance removes the need of disk crawling and index building, providing just-in-time analytics without *a priori* knowledge or pre-processing of the file systems. This is desirable in situations when the metadata indexes are not available, a query is not supported by the index, or query processing is only scarcely needed.

Using sampling for processing analytical queries is by no means new. Studies on sampling flat files, hashed files, and files generated by a relational database system (e.g., a B+-tree file) started more than 20 years ago - see survey [39] - and were followed by a myriad of work on database sampling for approximate query processing in decision support systems - see tutorials [4, 15, 22]. A wide variety of sampling techniques, e.g., simple random sampling [38], stratified [10], reservoir [48] and cluster sampling [11], have been used. Nonetheless, to the best of our knowledge, there has been no existing work on using sampling to support efficient aggregate and top-k query processing over a large hierarchical file system, i.e., one with numerous files organized in a complex folder structure (tree-like or directed acyclic graph).

Our main contributions are two-fold: (1) Glance consists of two algorithms, *FS_Agg* and *FS_TopK*, for the approximate processing of aggregate and top-k queries, respectively. For just-in-time analytics over very large file systems, we develop a random descent technique for unbiased aggregate estimations and a pruning-based technique for top-k query processing. (2) We study the specific characteristics of real-world file systems and derive the corresponding enhancements to our proposed techniques. In particular, according to the distribution of files in real-world file systems, we propose a high-level crawling technique to significantly reduce the error of query processing. Based on an analysis of accuracy and efficiency for the descent process, we propose a breadth-first implementation to reduce both error and overhead. We evaluate Glance over both real-world (e.g., NTFS, NFS, Plan 9) and synthetic file systems and find very promising results - e.g., 90% accuracy at 20% cost. Furthermore, we demonstrate that Glance is scalable to one billion of files and millions of directories.

We would like to note, however, that Glance also has its limitations - there are certain ill-formed file systems that malicious users could potentially construct so that Glance cannot effectively handle. While we plan to address security applications in future work, our argument of Glance being a practical system for just-in-time analytics is based upon the fact that these systems rarely exist in practice. For example, Glance cannot accurately answer aggregate queries if a large number of folders are hundreds of levels below root. Nonetheless, real-world file systems would have far smaller depth, making such a scenario unlikely to occur. Similarly, Glance cannot efficiently handle cases where all files have extremely close scores. This, however, is contradicted by the heavy-tailed distribution observed on most meta-data attributes in real-world file systems [2].

The rest of the paper is organized as follows. Section 2 presents the problem definition. In Section 3 and 4, we describe FS_Agg and FS_TopK for processing aggregate and top-k queries, respectively. The evaluation results are shown in Section 5. Section 6 reviews the related work, followed by the conclusion in Section 7.

2 Problem Statement

We now define the analytical queries, i.e., aggregate and top-k ones, which we focus on in this paper. The examples we list below will be used in the experimental evaluation for testing the performance of Glance.

Aggregate Queries: In general, aggregate queries are of the form *SELECT AGGR(T) FROM D WHERE Selection Condition*, where *D* is a file system or storage device, *T* is the target piece of information, which may be a metadata attribute (e.g., size, timestamp) of a file or a directory, *AGGR* is the aggregate function (e.g., COUNT, SUM, AVG), and *Selection Condition* specifies which files and/or directories are of interest. First, consider a system administrator who is interested in the total number of files in the system. In this case, the aggregate query that the administrator would like to issue can be expressed as:

Q1: SELECT COUNT(files) FROM filesystem;

Further, the administrator may be interested in knowing the total size of various types of document files, e.g.,

Q2: SELECT SUM(file.size) FROM filesystem WHERE file.extension IN { 'txt', 'doc'};

If the administrator wants to compute the average size of all exe files from user John, the query becomes:

Q3: SELECT AVG(file.size) FROM filesystem WHERE file.extension = 'exe' AND file.owner = 'John';

Aggregate queries can also be more complex - the following example shows a nested aggregate query for scientific computing applications. Suppose that each directory is corresponding to a sensor and contains a number of files corresponding to the sensor readings received at different time. A physicist may want to count the number of sensors that has received at least one reading during the last 12 hours, i.e.,

*Q4: SELECT COUNT(directories) FROM filesystem WHERE EXISTS (SELECT * FROM filesystem WHERE file.dirname = directory.name AND file.mtime BETWEEN (now − 12 hours) AND now);*

Top-k Queries: In this paper, we also consider top-k queries of the form *SELECT TOP k FROM D WHERE Selection Condition ORDER BY T DESCENDING/ASCENDING*, where T is the *scoring function* based on which the top-k files or directories are selected. For example, a system administrator may want to select the 100 largest files, i.e.,

Q5: SELECT TOP 100 files FROM filesystem ORDER BY file.size DESCENDING;

Another example is to find the ten most recently created directories that were modified yesterday, i.e.,

Q6: SELECT TOP 10 directories FROM filesystem WHERE directory.mtime BETWEEN (now − 24 hours) AND now ORDER BY directory.ctime DESCENDING;

We note that, to approximately answer a top-k query, one shall return a list of k items that share a large percentage of common ones with the precise top-k list.

Current operating systems and storage devices do not provide APIs which directly support the above-defined aggregate and top-k queries. The objective of just-in-time analytics can be stated as follows.

Problem Statement (Objective of Just-In-Time Analytics over File Systems): To enable the efficient approximate processing of aggregate and top-k queries over a file system by using the file/directory access APIs provided by the operating system.

To complete the problem statement, we need to determine how to measure the efficiency and accuracy of query processing. For the purpose of this paper, we measure the query efficiency in two metrics: 1) *query time*, i.e., the runtime of query processing, and 2) *query cost*, i.e., the ratio of the number of directories visited by Glance to that of crawling the file system (i.e., the total number of directories in the system). We assume that one disk access is required for reading a new directory. Thus, the query cost approximates the number of disk accesses required by Glance. The two metrics, query time and cost, are positively correlated - the higher the query cost is, more directories the algorithm has to sample, leading to a longer runtime.

While the efficiency measures are generic to both aggregate and top-k query processing, the measures for query accuracy are different. For aggregate queries, we define the query accuracy as the relative error of the approximate answer apx compared with the precise one ans - i.e., $|apx - ans|/|ans|$. For top-k queries, we define the accuracy as the percentage of items that are common in the approximate and precise top-k lists. The accuracy level required for approximate query processing depends on the intended application. For example, while scientific computing usually requires a small error, the above-mentioned surveillance application may simply need a ball-park figure to determine whether there is a significant amount of sensitive files in the system.

3 Aggregate Query Processing

In this section, we develop *FS_Agg*, our algorithm for processing aggregate queries. We first describe *FS_Agg_Basic*, a vanilla algorithm which illustrates our main idea of aggregate estimation without bias through a random descent process within a file system. Then, we describe two ideas to make the vanilla algorithm practical over very large file systems: *high-level crawling* leverages the special properties of a file system to reduce the standard error of estimation, and *breadth-first implementation* improves both accuracy and efficiency of query processing. Finally, we combine all three techniques to produce FS_Agg.

3.1 FS_Agg_Basic

A Random Descent Process: In general, the folder organization of a file system can be considered as a tree or a directed acyclic graph (DAG), depending on whether the file system allows hard links to the same file. The random descent process we are about to discuss can be applied to both cases with little change. For the ease of understanding, we first focus on the case of tree-like folder structure, and then discuss a simple extension to DAG at the end of this subsection.

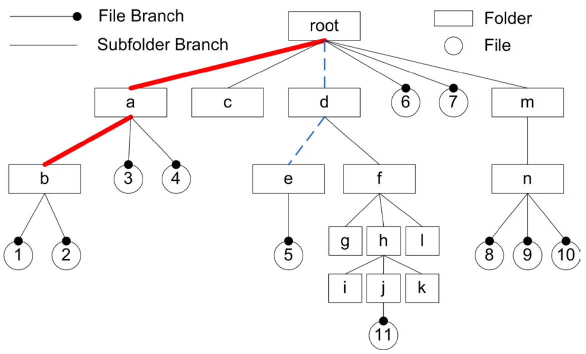


Figure 1: Random descents on a tree-like structure

Figure 1 depicts a tree structure with root corresponding to the root directory of a file system, which we shall use as a running example throughout the paper. One can see from the figure that there are two types of nodes in the tree: folders (directories) and files. A file is always a leaf node. The children of a folder consist of all subfolders and files in the folder. We refer to the branches coming out of a folder node as subfolder-branches and file-branches, respectively, according to their destination type. We refer to a folder with no subfolder-branches as a *leaf-folder*. Note that this differs from a leaf in the tree, which can be either a file or a folder containing neither subfolder nor file. The random descent process starts from the root and ends at a leaf-folder. At each node, we choose a subfolder branch of the node uniformly at random for further exploration. During the descent process, we evaluate all file branches encountered at each node along the path, and generate an aggregate estimation based on these file branches.

To make the idea more concrete, consider an example of estimating the COUNT of all files in the system. At the beginning of random descent, we access the root to obtain the number of its file- and subfolder-branches f_0 and s_0, respectively, and record them as our evaluation for the root. Then, we randomly choose a subfolder-branch for further descent, and repeat this process until we arrive at a folder with no subfolder. Suppose that the numbers we recorded during such a descent process are $f_0, s_0, f_1, s_1, \ldots, f_h, s_h$, where s_h = 0 because each descent ends at a leaf-folder. We estimate the COUNT of all files as

$$\tilde{n} = \sum_{i=0}^{h} \left(f_i \cdot \prod_{j=0}^{i-1} s_j \right), \qquad (1)$$

where $\prod_{j=0}^{i-1} s_j$ is assumed to be 1 when $i = 0$. Two examples of such a random descent process are marked in Figure 1 as red solid and blue dotted lines, respectively. The solid descent produces $\langle f_0, f_1, f_2 \rangle = \langle 2, 2, 2 \rangle$ and $\langle s_0, s_1, s_2 \rangle = \langle 4, 1, 0 \rangle$, leading to an estimation of 2 + 8 + 8 = 18. The dotted one produces $\langle f_0, f_1, f_2 \rangle = \langle 2, 0, 1 \rangle$ and $\langle s_0, s_1, s_2 \rangle = \langle 4, 2, 0 \rangle$, leading to an estimation of 2 + 0 + 8 = 10. The random descent process can be repeated multiple times (by restarting from the root) to produce a more accurate result (by taking the average of estimations generated by all descents).

Unbiasedness: Somewhat surprisingly, the estimation produced by each random descent process is completely *unbiased* - i.e., the expected value of the estimation is exactly equal to the total number of files in the system. To understand why, consider the total number of files at the i-th level (with root being Level 0) of the tree (e.g., Files 1 and 2 in Figure 1 are at Level 3), denoted by F_i. According to the definition of a tree, each i-level file belongs to one and only one folder at Level $i - 1$. For each $(i - 1)$-level folder v_{i-1}, let $|v_{i-1}|$ and $p(v_{i-1})$ be the number of (i-level) files in v_{i-1} and the probability for v_{i-1} to be reached in the random descent process, respectively. One can see that $|v_{i-1}|/p(v_{i-1})$ is an unbiased estimation for $F(i)$ because

$$E\left(\frac{|v_{i-1}|}{p(v_{i-1})}\right) = \sum_{v_{i-1}} \left(p(v_{i-1}) \cdot \frac{|v_{i-1}|}{p(v_{i-1})} \right) = F_i. \qquad (2)$$

With our design of the random descent process, the probability $p(v_{i-1})$ is

$$p(v_{i-1}) = \prod_{j=0}^{i-2} \frac{1}{s_j(v_{i-1})}, \qquad (3)$$

where $s_j(v_{i-1})$ is the number of subfolder-branches for each node encountered on the path from the root to v_{i-1}. Our estimation in (1) is essentially the sum of the unbiased estimations in (2) for all $i \in [1, m]$, where m is the maximum depth of a file. Thus, the estimation generated by the random descent is unbiased.

Processing of Aggregate Queries: While the above example is for estimating the COUNT of all files, the same random descent process can be used to process queries with other aggregate functions (e.g., SUM, AVG), with selection conditions (e.g., COUNT all files with extension '.JPG'), and in file systems with a DAG instead of tree structure. We now discuss these extensions. In particular, we shall show the only change required for all these extensions is on the computation of f_i.

SUM: For the COUNT query, we set f_i to the number of files in a folder. To process a SUM query over a file metadata attribute (e.g., file size), we simply set f_i as the SUM of such an attribute over all files in the folder (e.g., total size of all files). In the running example, consider the estimation of SUM of numbers shown on all files in Figure 1. The solid and dotted random walks will return $\langle f_0, f_1, f_2 \rangle = \langle 15, 7, 3 \rangle$ and $\langle 15, 0, 5 \rangle$, respectively, leading to the same estimation of 55. The unbiasedness of such an estimation follows in analogy from the COUNT case.

AVG: A simple way to process an AVG query is to estimate the corresponding SUM and COUNT respectively, and then compute AVG as SUM/COUNT. Note, however, that such an estimation is no longer unbiased, because the division of two unbiased estimations is not necessarily unbiased. While an unbiased AVG estimation may indeed be desired for certain applications, we have proved a negative result that it is impossible to answer an AVG query without bias unless one accesses the file system for almost as many as times as *crawling* the file system. We omit the detailed proof here due to the space limitation. Nonetheless, for practical purposes, estimating AVG as SUM/COUNT is in general very accurate, as we shall show in the experimental results.

Selection Conditions: To process a query with selection conditions, the only change required is, again, on the computation of f_i. Instead of evaluating f_i over all file branches of a folder, to answer a conditional query, we only evaluate f_i over the files that satisfy the selection conditions. For example, to answer a query SELECT COUNT(*) FROM Files WHERE file.extension = 'JPG', we should set f_i as the number of files under the current folder with extension JPG. Similarly, to answer "SUM($file\_size$) WHERE owner = John", we should set f_i to the SUM of sizes for all files (under the current folder) which belong to John. Due to the computation of f_i for conditional queries, the descent process may be *terminated early* to further reduce the cost of sampling. Again consider the query condition of (owner = John). If the random descent reaches a folder which cannot be accessed by John, then it can terminate immediately because any deeper descent can only return f_i = 0, leading to no change in the estimation.

Extension to DAG Structure: Finally, for a file system featuring a DAG (instead of tree) structure, we again only need to change the computation of f_i. Almost all DAG-enabled file systems (e.g., ext2, ext3, NTFS) provide a *reference count* for each file which indicates the number of links in the DAG that point to the file[1]. For a file with r links, if we use the original algorithm discussed above, then the file will be counted r times in the estimation. Thus, we should discount its impact on each estimation with a factor of $1/r$. For example, if the query being processed is the COUNT of all files, then we should compute $f_i = \sum_{f \in F}(1/r(f))$, where F is the set of files under the current folder, and $r(f)$ is the number of links to each file f. Similarly, to estimate the SUM of all file sizes, we should compute $f_i = \sum_{f \in F}(size(f)/r(f))$, where $size(f)$ is the file size of file f. One can see that with this discount factor, we maintain an unbiased estimation over a DAG file system structure.

3.2 Disadvantages of FS_Agg_Basic

While the estimations generated by FS_Agg_Basic is unbiased for SUM and COUNT queries, it is important to understand that the error of an estimation comes from not only bias but also variance (i.e., standard error). A problem of FS_Agg_Basic is that it may produce a high estimation variance for file systems with an undesired distribution of files, as illustrated by the following theorem:

Theorem 1. *The variance of estimation produced by a random descent on the number of h-level files F_h is*

$$\sigma(h)^2 = \left(\sum_{v \in L_{h-1}} (|v|^2 \cdot \prod_{j=0}^{h-2} s_j(v)) \right) - F_h^2. \qquad (4)$$

where L_{h-1} is the set of all folders at Level $h-1$, $|v|$ is the number of files in a folder v, and $s_j(v)$ is the number of subfolders for the Level-j node on the path from the root to v.

Proof. Consider an $(h-1)$-level folder v. If the random descent reaches v, then the estimation it produces for the number of h-level files is $|v|/p(v)$, where $p(v)$ is the probability for the random descent to reach v. Let $\delta(h)$ be the probability that a random descent terminates

[1] In ext2 and ext3, for example, the system provides the number of hard links for each file. Note that for soft links, we can simply ignore them during the descent process. Thus, they bear no impact on the final estimation.

early before reaching a Level-$(h-1)$ folder. Since each random descent reaches at most one Level-$(h-1)$ folder, the estimation variance for F_h is

$$\sigma(h)^2 = \delta(h)\cdot F_h^2 + \sum_{v\in L_{h-1}} p(v)\cdot(\frac{|v|}{p(v)} - F_h)^2 \quad (5)$$

$$= \delta(h)\cdot F_h^2 + \sum_{v\in L_{h-1}} \left(\frac{|v|^2}{p(v)} - 2|v|F_h + p(v)\cdot F_h^2\right) \quad (6)$$

$$= \left(\sum_{v\in L_{h-1}} \frac{|v|^2}{p(v)}\right) - F_h^2 \quad (7)$$

Since $p(v) = 1/\prod_{j=0}^{h-2} s_j(v)$, the theorem is proved. □

One can see from the theorem that the existence of two types of folders may lead to an extremely high estimation variance: One type is *high-level leaf-folders* (i.e., "shallow" folders with no subfolders). Folder c in Figure 1 is an example. To understand why such folders lead to a high variance, consider (7) in the proof of Theorem 1. Note that for a large h, a high-level leaf-folder (above Level-$(h-1)$) reduces $\sum_{v\in L_{h-1}} p(v)$ because once a random descent reaches such a folder, it will not continue to retrieve any file in Level-h (e.g., Folder c in Figure 1 stops further descents for h = 3 or 4). As a result, the first item in (7) becomes higher, increasing the estimation variance. For example, after removing Folder c from Figure 1, the estimation variance for the number of files on Level 3 can be reduced from 24 to 9.

The other type of "ill-conditioned" folders are those *deep-level folders* which reside at much lower levels than others (i.e., with an extremely large h). An example is Folder j in Figure 1. The key problem arising from such a folder is that the probability for it to selected is usually extremely small, leading to an estimation much larger than the real value if the folder happens to be selected. As shown in Theorem 1, a larger h leads to a higher $\prod s_j(v)$, which in turn leads to a higher variance. For example, Folder j in Figure 1 has $\prod s_j(v) = 4\times 2\times 3\times 3 = 72$, leading to a estimation variance of $72 - 1 = 71$ for the number of files on Level 5 (which has a real value of 1).

3.3 FS_Agg

To reduce the estimation variance, we propose high-level crawling and breadth-first descent to address the two above-described problems on estimation variance, high-level leaf-folders and deep-level folders, respectively. Also, we shall discuss how the variance generated by FS_Agg can be estimated in practice, effectively producing a confidence interval for the aggregate query answer.

High-Level Crawling is designed to eliminate the negative impact of high-level leaf-folders on estimation variance. The main idea of high-level crawling is to access all folders in the highest i levels of the tree - by following all subfolder-branches of folders accessed on or above Level-$(i-1)$. Then, the final estimation becomes an aggregate of two components: the precise value over the highest i levels and the estimated value (produced by random descents) over files below Level-i. One can see from the design of high-level crawling that now leaf-folders in the first i levels no longer reduce $p(v)$ for folders v below Level-i (and therefore no longer adversely affect the estimation variance). Formally, we have the following theorem[2] which demonstrates the effectiveness of high-level crawling on reducing the estimation variance:

Theorem 2. *If r_0 out of r folders crawled from the first i levels are leaf-folders, then the estimation variance produced by a random descent for the number of Level-h files F_h satisfies*

$$\sigma_{\text{HLC}}(h)^2 \leq \frac{(r-r_0)\cdot\sigma_h^2 - r_0\cdot F_h^2}{r}. \quad (8)$$

According to this theorem, if we apply high-level crawling over the first level in Figure 1, then the estimation variance for the number of files on Level 3 is at most $(3\cdot 24 - 1\cdot 36)/4 = 9$. Recall from Section 4.2 that the variance of estimation after removing Folder c (the only leaf-folder at the first level) is exactly 9. Thus, the bound in Theorem 2 is tight in this case.

Breadth-First Descent is designed to bring two advantages over FS_Agg_Basic: variance reduction and runtime improvement, which we shall explain as follows.
Variance Reduction: breadth-first descent starts from the root of the tree. Then, at any level of the tree, it generates a set of folders to access at the next level by randomly selecting from subfolders of all folders it accesses at the currently level. Note that any random selection process would work - as long as we know the probability for a folder to be selected, we can answer aggregate queries without bias in the same way as the original random descent process. For example, to COUNT the number of all files in the system, an unbiased estimation of the total number of files at Level i is the SUM of $|v_{i-1}|/p(v_{i-1})$ for all Level-$(i-1)$ folders v_{i-1} accessed by the breadth-first implementation, where $|v_{i-1}|$ and $p(v_{i-1})$ are the number of file-branches and the probability of selection for v_{i-1}, respectively.

We use the following random selection process in Glance: Consider a folder accessed at the current level which has n_0 subfolders. From these n_0 subfolders, we sample without replacement $\min(n_0, \max(p_{\text{sel}}\cdot$

[2] In the rest part of the paper, we do not include the proof of theorems due to the space limitation.

$n_0, s_{\min}))$ ones for access at the next level. Here $p_{\text{sel}} \in (0, 1]$ (where sel stands for selection) represents the probability of which a subfolder will be selected for sampling, and $s_{\min} \geq 1$ states the minimum number of subfolders that will be sampled. Both p_{sel} and $s_{\min}$ are user-defined parameters, the settings for which we shall further discuss in the experiments section based on characteristics of real-world file systems.

Compared with the original random descent design, this breadth-first random selection process significantly increases the selection probability for a deep folder. Recall that with the original design, while drilling down one level down the tree, the selection probability can decrease rapidly by a factor of the fan-out (i.e., the number of subfolders) of the current folder. With breadth-first descent, on the other hand, the decrease is limited to at most a factor of $1/p_{\text{sel}}$, which can be much smaller than the fanout when p_{sel} is reasonably high (e.g., =0.5 as we shall suggest in the experiments section). As a result, the estimation generated by a deep folder becomes much smaller. Formally, we have the following theorem.

Theorem 3. *With breadth-first descent, the variance of estimation on the number of h-level files F_h satisfies*

$$\sigma_{\text{BFS}}(h)^2 \leq \left(\sum_{v \in L_{h-1}} \frac{|v|^2}{p_{\text{sel}}^{h-1}} \right) - F_h^2. \qquad (9)$$

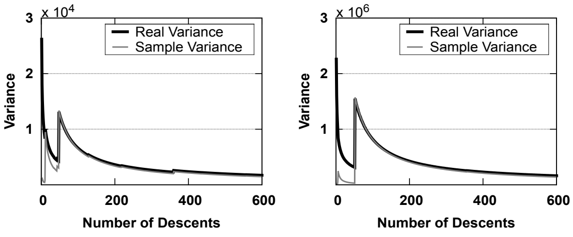
One can see from a comparison with Theorem 1 that the factor of $\prod s_j(v)$ in the original variance, which can grow to an extremely large value, is now replaced by $1/p_{\text{sel}}^{h-1}$ which can be better controlled by the Glance system to remain at a low level even when h is large.

Runtime Improvement: In the original design of FS_Agg_Basic, random descent has to be performed multiple times to reduce the estimation variance. Such multiple descents are very likely to access the same folders, especially the high-level ones. While one can leverage the history of hard-drive accesses by caching all historic accesses in memory, such repeated accesses can still take significant CPU time for in-memory look up. The breadth-first design, on the other hand, ensures that each folder is accessed at most once, reducing the runtime overhead of the Glance system.

Variance Produced by FS_Agg: An important issue for applying FS_Agg in practice is how one can estimate the error of approximate query answers it produces. Since FS_Agg generates unbiased answers for SUM and COUNT queries, the key enabling factor for error estimation here is an accurate computation of the variance. One can see from Theorem 3 that variance depends on the specific structure of the file system, in particular the distribution of selection probability p_{sel} for different folders. Since our sampling-based algorithm does not have a global view of the hierarchical structure, it cannot precisely compute the variance.

Fortunately, the variance can still be accurately *approximated* in practice. To understand how, consider first the depth-first descents used in FS_Agg_Basic. Each descent returns an independent aggregate estimation, while the average for multiple descents becomes the final approximate query answer. Let $\tilde{q}_1, \ldots, \tilde{q}_h$ be the independent estimations and $\tilde{q} = (\sum \tilde{q}_i)/h$ be the final answer. A simple method of variance approximation is to compute $var(\tilde{q}_1, \ldots, \tilde{q}_h)/h$, where $var(\cdot)$ is the variance of independent estimations returned by the descents. Note that if we consider a population consisting of estimations generated by all possible descents, then $\tilde{q}_1, \ldots, \tilde{q}_h$ form a sample of the population. As such, the variance computation is approximating the population variance by sample variance, which are asymptotically equal (for an increasing number of descents).

We conducted extensive experiments described in Section 5 to verify the accuracy of such an approximation. Figure 2 shows two examples for counting the total number of files in an NTFS and a Plan 9 file system, respectively. Observe from the figure that the real variance oscillates in the beginning of descents. For example, we observe at least one spike on each file system within the first 100 descents. Such a spike occurs when one descent happens to end with a deep-level file which returns an extremely large estimation, and is very likely to happen with our sampling-based technique. Nonetheless, note that the real variance converges to a small value when the number of descents is sufficiently large (e.g., > 400). Also note that for two file systems after a small number of descents (about 50), the sample variance $var(\tilde{q}_1, \ldots, \tilde{q}_h)/h$ becomes an extremely accurate approximation for the real (population) variance (overlapping shown in Figure 2), even during the spikes. One can thereby derive an accurate confidence interval for the query answer produced by FS_Agg_Basic.



(a) NTFS (b) Plan 9

Figure 2: Variance approximation for (a) an NTFS file system and (b) a Plan 9 system. Real and sample variances are overlapped when the number of descents is sufficiently large.

While FS_Agg no longer performs individual depth-first descents, the idea of using sample variance to approximate population variance still applies. In particular, note that for any given level, say Level-i, of the tree structure, each folder randomly chosen by FS_Agg at Level-$(i-1)$ produces an independent, unbiased, estimation for SUM or COUNT aggregate over all files in Level-i. Thus, the variance for an aggregate query answer over Level-i can be approximated based on the variance of estimations generated by the individual folders. The variance of final SUM or COUNT query answer (over the entire file system) can then be approximated by the SUM of variances for all levels.

4 Top-k Query Processing

Recall that for a given file system, a top-k query is defined by two elements: the *scoring function* and the *selection conditions*. Without loss of generality, we consider a top-k query which selects k files (directories) with the *highest* scores. For the sake of simplicity, we focus on top-k queries without selection conditions, and consider a tree-like structure of the file system. The extensions to top-k queries with selection conditions and file systems with DAG structures follow in analogy from the same extensions for FS_Agg.

4.1 Main Idea

A simple way to answer a top-k query is to access every directory to find the k files with the highest scores. The objective of FS_TopK is to generate an approximate top-k list with far fewer hard-drive accesses. To do so, FS_TopK consists of the following three steps. We shall describe the details of these steps in the next subsection.

1. *A Lower-Bound Estimation:* The first step uses a random descent similar to FS_Agg to generate an approximate lower bound on the k-th highest score over the entire file system (i.e., among files that satisfy the selection conditions specified in the query).
2. *Highest-Score Estimations and Tree Pruning:* In the second step, we prune the tree structure of the file system according to the lower bound generated in Step 1. In particular, for each subtree, we use the results of descents to generate an upper-bound estimate on the highest score of all files in the subtree. If the estimation is smaller than the lower bound from Step 1, we remove the subtree from search space because it is unlikely to contain a top-k file. Note that in order for such a pruning process to have a low false negative rate - i.e., not to falsely remove a large number of real top-k files, a key assumption we are making here is the "locality" of scores - i.e., files with similar scores are likely to co-locate in the same directory or close by in the tree structure. Intuitively, the files in a directory are likely to have similar creation and update times. In some cases (e.g., images in the "My Pictures" directory, and outputs from a simulation program), the files will likely have similar sizes too. Note that the strength of this locality is heavily dependent on the type of the query and the semantics of the file system on which the query is running. We plan to investigate this issue as part of the future work.
3. *Crawling of the Selected Tree:* Finally, we crawl the remaining search space - i.e., the selected tree - by accessing every folder in it to locate the top-k files as the query answer. Such an answer is approximate because some real top-k files might exist in the selected subtrees, albeit with a small probability, as we shall show in the experimental results.

In the running example, consider a query for the top-3 files with the highest numbers shown in Figure 1. Suppose that Step 1 generates a (conservative) lower bound of 8, and the highest scores estimated in Step 2 for subtrees with roots a, c, d, and m are 5, -1 (i.e., no file), 7, and 15, respectively - the details of these estimations will be discussed shortly. Then, the pruning step will remove the subtrees with roots a, c, and d, because their estimated highest scores are lower than the lower bound of 8. Thus, the final crawling step only needs to access the subtree with root of a. In this example, the algorithm would return the files identified as 8, 9, and 10, locating two top-3 files while crawling only a small fraction of the tree. Note that the file with the highest number 11 could not be located here because the pruning step removes the subtree with root of d.

4.2 Detailed Design

The design of FS_TopK is built upon a hypothesis that the highest scores estimated in Step 2, when compared with the lower bound estimated in Step 1, can prune a large portion of the tree, significantly reducing the overhead of crawling in Step 3. In the following, we first describe the estimations of the lower bound and the highest scores in Steps 1 and 2, and then discuss the validity of the hypothesis for various types of scoring functions.

Both estimations in the two steps can be made from the *order statistics* [20] of files retrieved by the random descent process in FS_Agg. The reason is that both estimations are essentially on the order statistics of the population (i.e., all files in the system) - The lower bound in Step 1 is the k-th largest order statistics of all files, while the highest scores are on the largest order statistics of the subtrees. We refer readers to [20] for details of how the order statistics of a sample can be used to estimate that of the population and how accurate such an estimation is.

While sampling for order statistics is a problem of its own right, for the purpose of this paper, we consider the following simple approach which, according to our experiments over real-world file systems, suffices for answering top-k queries accurately and efficiently over almost all tested systems: For the lower-bound estimation in Step 1, we use the sample quantile as an estimation of the population quantile. For example, to estimate the 100-th largest score of a system with $10,000$ files, we use the largest score of a 100-file sample as an estimation. Our tests show that for many practical scoring functions (which usually have a positive skew, as we shall discuss below), the result serves as a conservative lower bound desired by FS_TopK. For the highest-score estimation in Step 2, we simply compute $\gamma \cdot \max(\text{sample scores})$, where γ is a constant correction parameter. The setting of γ captures a tradeoff between the crawling cost and the chances of finding top-k files - when a larger γ is selected, less number of the subtrees are likely be removed.

We now discuss when the hypothesis of heavy pruning is valid and when it is not. Ideally, two conditions should be satisfied for the hypothesis to hold: (1) If a subtree includes a top-k file, then it should include a (relatively) large number of highly scored files, in order for the sampling process (in Step 2) to capture one (and to thereby produce a highest-score estimation that surpasses the lower bound) with a small query cost. And (2) on the other hand, most subtrees (which do not include a top-k file) should have a maximum score significantly lower than the k-th highest score. This way, a large number of subtrees can be pruned to improve the efficiency of top-k query processing. In general, one can easily construct a scoring function that satisfy both or neither of the above two conditions. We focus on a special class of scoring functions: those following a heavy-tailed distributions (i.e., its cumulative distribution function $F(\cdot)$ satisfies $\lim_{x\to\infty} e^{\lambda x}(1 - F(x)) = \infty$ for all $\lambda > 0$). Existing studies on real-world file system traces showed that many file/directory metadata attributes, which are commonly used as scoring functions, belong to this category [2]. For example, the distributions of file size, last modified time, creation time, etc., in the entire file system or in a particular subtree are likely to have a heavy tail on one or both extremes of the distribution.

A key intuition here is that scoring functions defined as such attribute values (e.g., finding the top-k files with the maximum sizes or the latest modified time) usually satisfy both conditions: First, because of the long tail, a subtree which includes a top-k scored file is likely to include many other highly scored files as well. Second, since the vast majority of subtrees have their maximum scores significantly smaller than the top-k lower bound, the pruning process is likely to be effective with such a scoring function.

We would also like to point out an "opposite" class of scoring functions for which the pruning process is not effective: the inverse of the above scoring functions - e.g., the top-k files with the smallest sizes. Such a scoring function, when used in a top-k query, selects k files from the "crowded" light-tailed side of the distribution. The pruning is less likely to be effective because many other folders may have files with similar scores, violating the second condition stated above. Fortunately, asking for top-k smallest files is not particularly useful in practice, also because of the fact that it selects from the crowded side - e.g., the answer is likely to be a large number of empty files.

5 Implementation and Evaluation

5.1 Implementation

We implemented Glance, including all three algorithms (FS_Agg_Basic, FS_Agg and FS_TopK) in 1,600 lines of C code in Linux. We also built and used a simulator in Matlab to complete a large number of tests within a short period of time. While the implementation was built upon the ext3 file system, the algorithms are generic to any hierarchical file system and the current implementation can be easily ported to other platforms, e.g., Windows and Mac OS. FS_Agg_Basic has only one parameter: the number of descents. FS_Agg has three parameters: the selection probability p_{sel}, the minimum number of selections $s_{\min}$ and the number of (highest) levels for crawling h. Our default parameter settings are p_{sel} = 50%, $s_{\min}$ = 3, and h = 4. We also tested with other combinations of parameter settings. FS_TopK has one additional parameter, the (estimation) enlargement ratio γ. The setting of γ depends on the query to be answered, which shall be explained later.

5.2 Experiment Setup

Test Platform: We ran all experiments on Linux machines with Intel Core 2 Duo processor, 4GB RAM, and 1TB Samsung 7200RPM hard drive. Unless otherwise specified, we ran each experiment for five times and reported the averages.

Windows File Systems: The Microsoft traces [2] includes the snapshots of around 63,000 file systems, 80% of which are NTFS and the rest are FAT. To test Glance over file systems with a wide range of sizes, we first selected from the traces two file systems, $m100K$ and $m1M$ (the first 'm' stands for Microsoft trace), which are the largest file systems with less than 100K and 1M files, respectively. Specifically, $m100K$ has 99,985 files and 16,013 directories, and $m1M$ has 998,472 files and 106,892 directories. We also tested the largest system in the trace, $m10M$, which has the maximum number of files (9,496,510) and directories (789,097). We put together the largest 33 file systems in the trace to obtain $m100M$ that contains over 100M files and 7M directories. In order to evaluate next-generation billion-level file systems for which there are no available traces, we chose to replicate $m100M$ for 10 times to create $m1B$ with over 1 billion files and 70M directories. While a similar scale-up approach has been used in the literature [26,49], we would like to note that the duplication-filled system may exhibit different properties from a real system with 100M or 1B files. As part of future work, we shall evaluate our techniques in real-world billion-level file systems.

Plan 9 File Systems: Plan 9 is a distributed file system developed and used at the Bell Labs [41, 42]. We replayed the trace data collected on two central file servers *bootes* and *emelie*, to obtain two file systems, pb (for *bootes*) and pe (for *emelie*), each of which has over 2M files and 70-80K directories.

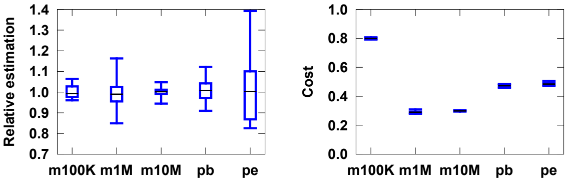
NFS: Here we used the Harvard trace [21, 45] that consists of workloads on NFS servers. The replay of one-day trace created about 1,500 directories and 20K files. Again, we scaled up the one-day system to a larger file system *nfs* (2.3M files and 137K folders), using the above-mentioned approach.

Synthetic File Systems: To conduct a more comprehensive set of experiments on file systems with different file and directory counts, we used *Impressions* [1] to generate a set of synthetic file systems. By adjusting the file count and the (expected) number of files per directory, we used Impressions to generate three file systems, $i10K$, $i100K$, and $i1M$ (here 'i' stands for Impressions), with file counts 10K, 100K, and 1M, and directory counts 1K, 10K, and 100K, respectively.

5.3 Aggregate Queries

We first considered Q1 discussed in Section 2, i.e., the total number of files in the system. To provide a more intuitive understanding of query accuracy (than the arguably abstract measure of relative error), we used the Matlab simulator (for quick simulation) to generate a box plot (Figure 3(a)) of estimations and overhead produced by Glance on Q1 over five file systems, m100K to m10M, pb and pe. Remember as defined in Section 2, the query cost (in Figure 3(b) and the following figures) is the ratio between the number of directories visited by Glance and that by file-system crawling. One can see that Glance consistently generates accurate query answers, e.g., for m10M, sampling 30% of directories produces an answer with 2% average error. While there are outliers, the number of outliers is small and their errors never exceed 7%.

We also evaluated Glance with other file systems and varied the input parameter settings. This test was conducted on the Linux and ext3 implementation, and so were the following tests on aggregate queries. In this test, we varied the minimum number of selections $s_{\min}$ from 3 to 6, the number of crawled levels h from 3 to 5, and set the selection probability as p_{sel} = 50% (i.e., half of the subfolders will be selected if the amount is more than $s_{\min}$). Figure 4 shows the query accuracy and cost on the eleven file systems we tested. For all file systems, Glance was able to produce very accurate answers (with <10% relative error) when crawling four or more levels (i.e., $h \geq 4$). Also note from Figure 4 that the performance of Glance is less dependent on the type of the file system than its size - it achieves over 90% accuracy for NFS, Plan 9, and NTFS (m10M to m1B). Depending on the individual file systems, the cost ranges from less than 12% of crawling for large systems with 1B files and 80% for the small 100K system. The algorithm scales very well to large file systems e.g., m100M and m1B - the relative error is only 1-3% when Glance accesses only 10-20% of all directories. For m1B, the combination of p_{sel} = 50%, $s_{\min}$ = 3 and h = 4 produces 99% accuracy with very little cost (12%).



(a) Accuracy

(b) Cost

Figure 3: Box plots of accuracy and cost of 100 trials

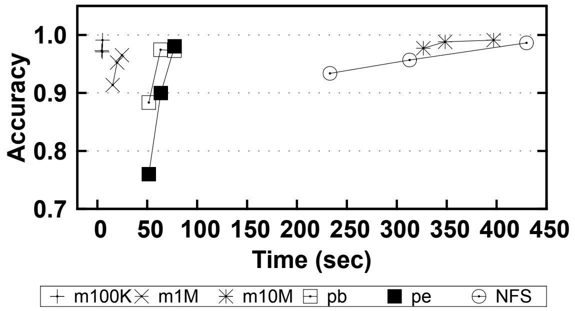


Figure 5: Query accuracy vs. run time in seconds. Three points of each line (from left to right) represent h of 3, 4, and 5, respectively.

Figure 5 illustrates the runtimes (in seconds) for aggregate queries. The absolute runtime depends heavily on the size of the file system, e.g., seconds for m100K, several minutes for nfs (2.3M files), and 1.2 hours for m100M (not shown in the figure). Note that in this paper we only used a single hard drive; parallel IO to multiple hard drives (e.g., RAID) will be able to utilize the aggregate bandwidth to further improve the performance. As the value of h increases, the query runs slightly longer

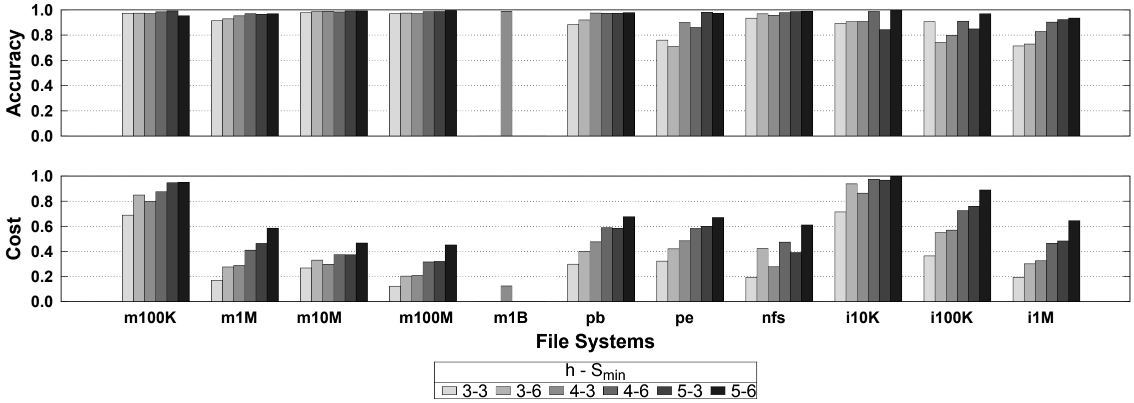
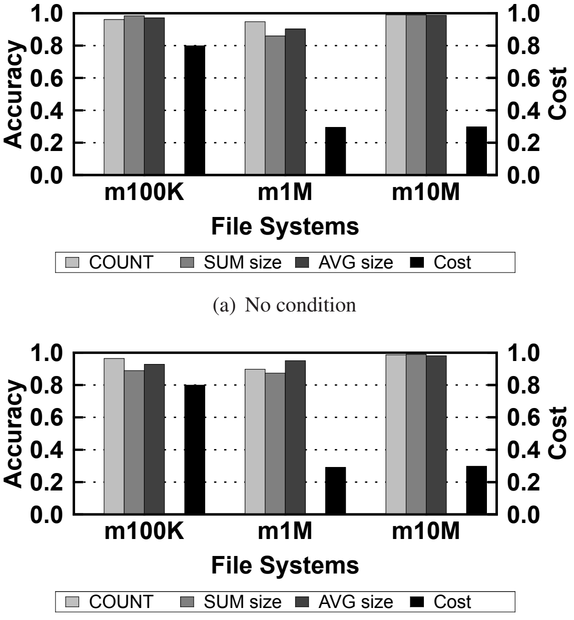


Figure 4: Accuracy and cost of aggregate queries under different settings of the input parameters. Label 3-3 stands for h of 3 and $s_{\min}$ of 3, 3-6 for h of 3 and $s_{\min}$ of 6, etc., while p_{sel} is 50% for all cases.

but the accuracy improves by about 10% for pb and 20% for pe. The accuracy improvements for m10M and nfs are smaller. The value of $s_{\min}$ is 3 in this test.



(a) No condition

(b) With condition

Figure 6: Accuracy and cost of queries

We also considered other aggregate queries with various aggregate functions and with/without selection conditions. Figure 6(a) presents the accuracy and cost of evaluating the SUM and AVG of file sizes for all files in the system, while Figure 6(b) depicts the same for *exe* files. We included in both figures the accuracy of COUNT because AVG is calculated as SUM/COUNT. Both SUM and AVG queries receive very accurate answers, e.g., only 2% relative error for m10M with or without the selection condition of '*.exe*'. The query costs are moderate for large systems - 30% for m1M and m10M (higher for the small system m100K). We also tested SUM and AVG queries with other selection conditions (e.g., file type = '*.dll* ') and found similar results.

5.4 Top-k Queries

To evaluate the performance of FS_TopK, we considered both Q5 and Q6 discussed in Section 2. For Q5, i.e., the k largest files, we tested Glance over five file systems, with k being 50 or 100. One can see from the results depicted in Figure 7 that, in all but one case (m1M), Glance is capable of locating at least 50% of all top-k files (for pb, more than 95% are located). Meanwhile, the cost is as little as 4% of crawling (for m10M). Figure 8 presents the runtimes of the top-k queries, where one can see that similar to aggregate queries, the runtime is correlated to the size of the file system - the queries take only a few seconds for small file systems, and up to ten minutes for large systems (e.g., m10M).

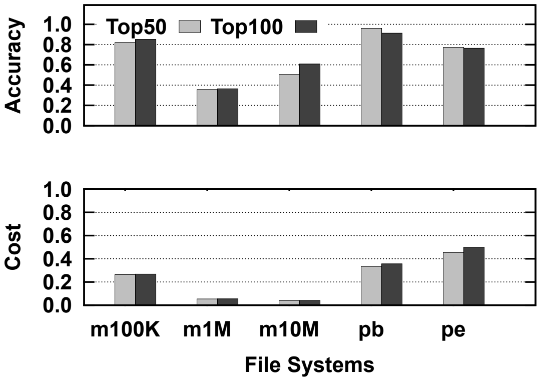


Figure 7: Accuracy and cost of Top-k queries on file size

Figure 9 presents the query accuracy and cost for Top-k queries on file size, when γ varies from 1, 5, 10, to 100,000. The trend is clear - the query cost increases as γ does, because a higher value of γ is to scale the highest-score estimation up to a larger degree, that is, to crawl a

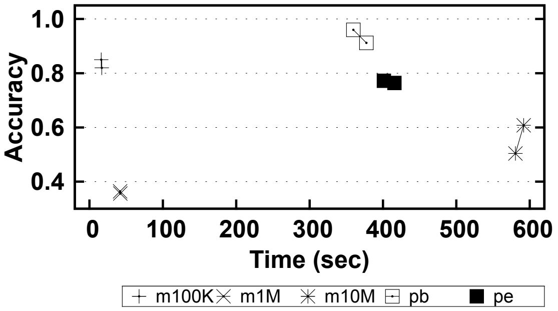


Figure 8: Top-k query accuracy vs. run time in seconds. The first point of each line stands for top-50 and the second for top-100.

larger portion of the file system. Fortunately, a moderate γ of 5 and 10 presents a good tradeoff point - achieving a reasonable accuracy without incurring too much cost.

We also tested Q6, i.e., the k most recently modified files over m100K, m1M, and pb. The results are shown in Figure 10. One can see that Glance is capable of locating more than 90% of top-k files for pb, and about 60% for m100K and m1M. The cost, meanwhile, is 28% of crawling for m100K, 1% for m1M, and 36% for pb.

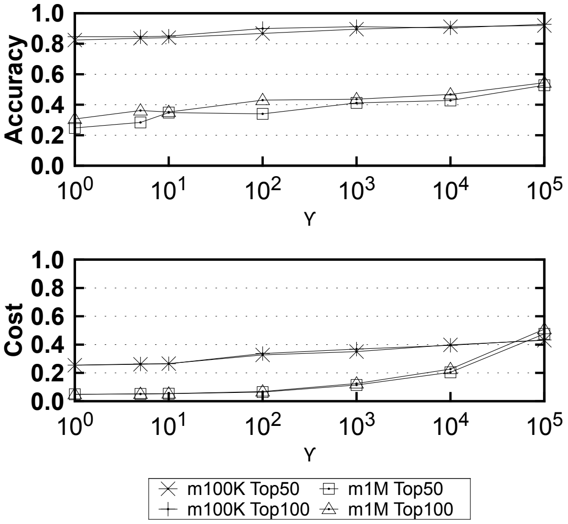


Figure 9: Query accuracy and cost when varying γ

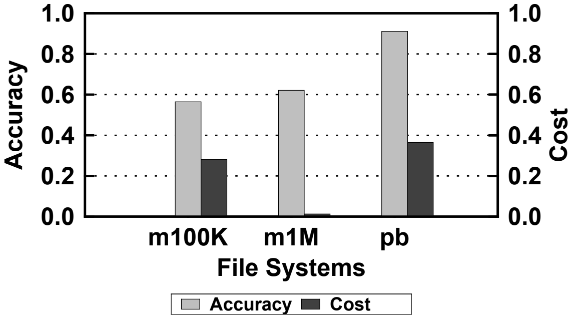


Figure 10: Top-k queries on file time

6 Related Work

Metadata query on file systems: Prior research on file-system metadata query [26, 32] has extensively focused on databases, which utilizes indexes on file metadata. However, the results [26,31,32] reviewed the inefficiency of this paradigm due to metadata locality and distribution skewness in large file systems. To solve this problem, Spyglass [30, 32], SmartStore [26], and Magellan [31] utilize multi-demensional structures (e.g., K-D trees and R-trees) to build indexes upon subtree partitions or semantic groups. SmartStore attempts to reorganize the files based on their metadata semantics. Conversely, Glance avoids any file-specific optimizations, aiming instead to maintain file system agnosticism. It works seamlessly with the tree structure of a file system and avoids the time and space overheads from building and maintaining the metadata indexes.

Comparison with Database Sampling: Traditionally database sampling has been used to reduce the cost of retrieving data from a DBMS. Random sampling mechanisms have been extensively studied [4, 6, 9, 12, 14, 15, 22, 34]. Applications of random sampling include estimation methodologies for histograms and approximate query processing (see tutorial in [15]). However, these techniques do not apply when there is no direct random access to all elements of interest - e.g., in a file system, where there is no complete list of all files/directories.

Another particularly relevant topic is the sampling of hidden web databases [8,24,25,28], for which a random descent process has been used to construct queries issued over the web interfaces of these databases [16–19]. While both these techniques and Glance use random descents, a unique challenge for sampling a file system is its much more complex distribution of files. If we consider a hidden database in the context of a file system, then all files (i.e., tuples) appear under folders with no subfolders. Thus, the complex distribution of files in a file system calls for a different sampling technique which we present in the paper .

Top-k Query Processing: Top-k query processing has been extensively studied over both databases (e.g., see a recent survey [27]) and file systems [3,7,26,32]. For file systems, a popular application is to locate the top-k most frequent (or space-consuming) files/blocks for redundancy detection and removal. For example, Lillibridge et al. [33] proposed the construction of an in-memory sparse index to compare an incoming block against a few (most similar) previously stored blocks for duplicate detections (which can be understood as a top-k query with a scoring function of similarity). Top-k query processing has also been discussed in other index-building techniques, e.g., in Spyglass [32] and SmartStore [26].

7 Discussion

At present, Glance takes several pre-defined parameters as the inputs and needs to complete the execution in whole. That is, Glance is not an any-time algorithm and cannot be stopped in the middle of the execution, because our current approach relies on a complete sample to reduce query variance and achieve high accuracy. One limitation of this approach is that its runtime over an alien file system is unknown in advance, making it unsuitable for the applications with absolute time constraints. For example, a border patrol agent may need to count the amount of encrypted files in a traveler's hard drive, in order to determine whether the traveler could be transporting sensitive documents across the border [13, 44]. In this case, the agent must make a fast decision as the amount of time each traveler can be detained for is extremely limited. We envision that in the future Glance shall offer a time-out knob that a user can use to decide the query time over a file system. This calls for new algorithms that allow Glance get smarter - be predictive about the run-time and self-adjust the work flow based on the real-time requirements.

Glance currently employs a "static" strategy over file systems and queries, i.e., it does not modify its techniques and traversals for a query. A dynamic approach is attractive because in that case Glance would be able to adjust the algorithms and parameters depending on the current query and file system. New sampling techniques, e.g., stratified and weighted sampling, shall be investigated to further improve query accuracy on large file systems. The semantic knowledge of a file system can also help in this approach. For example, most images can be found in a special directory, e.g. "/User/Pictures/" in MacOS X, or "\Documents and Settings\User\My Documents\My Pictures\" in Windows XP.

Glance shall also leverage the results from the previous queries to significantly expedite the future ones, which is beneficial in situations when the workload is a set of queries that are executed very infrequently. The basic idea is to store the previous estimations over parts (e.g., subtrees) of the file system, and utilize the history to limit the search space to the previously unexplored part of the file system, unless it determines that the history is obsolete (e.g., according to a pre-defined validity period). Note that the history shall be continuously updated to include newly discovered directories and to update the existing estimations.

8 Conclusion

In this paper we have initiated an investigation of just-in-time analytics over a large-scale file system through its tree- or DAG-like structure. We proposed a random descent technique to produce unbiased estimations for SUM and COUNT queries and accurate estimations for other aggregate queries, and a pruning-based technique for the approximate processing of top-k queries. We proposed two improvements, high-level crawling and breadth-first descent, and described a comprehensive set of experiments which demonstrate the effectiveness of our approach over real-world file systems.

9 Acknowledgments

We thank the anonymous reviewers and our shepherd John Bent for their excellent comments that helped improve the quality of this paper. We also thank Hong Jiang and Yifeng Zhu for their help on replaying the NFS trace, and Ron Chiang for his help on the artwork. This work was supported by the NSF grants OCI-0937875, OCI-0937947, IIS-0845644, CCF-0852674, CNS-0852673, and CNS-0915834.

References

[1] AGRAWAL, N., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. Generating realistic impressions for file-system benchmarking. *ACM Transactions on Storage (TOS) 5*, 4 (2009), 1–30.

[2] AGRAWAL, N., BOLOSKY, W., DOUCEUR, J., AND LORCH, J. A five-year study of file-system metadata. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies* (2007), pp. 31–45.

[3] AMES, S., GOKHALE, M., AND MALTZAHN, C. Design and implementation of a metadata-rich file system. Tech. Rep. UCSC-SOE-10-07, University of California, Santa Cruz, 2010.

[4] BARBARA, D., DUMOUCHEL, W., FALOUTSOS, C., HAAS, P., HELLERSTEIN, J., IOANNIDIS, Y., JAGADISH, H., JOHNSON, T., NG, R., POOSALA, V., ET AL. The New Jersey data reduction report. *IEEE Data Eng. Bull. 20*, 4 (1997), 3–45.

[5] BEAGLE. *http://beagle-project.org/*.

[6] BETHEL, J. Sample allocation in multivariate surveys. *Survey methodology 15*, 1 (1989), 47–57.

[7] BRANDT, S., MALTZAHN, C., POLYZOTIS, N., AND TAN, W.-C. Fusing data management services with file systems. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage (PDSW '09)* (New York, NY, USA, 2009), ACM, pp. 42–46.

[8] CALLAN, J., AND CONNELL, M. Query-based sampling of text databases. *ACM Trans. Inf. Syst. 19* (April 2001), 97–130.

[9] CAUSEY, B. Computational aspects of optimal allocation in multivariate stratified sampling. *SIAM Journal on Scientific and Statistical Computing 4* (1983), 322.

[10] CHAUDHURI, S., DAS, G., AND NARASAYYA, V. Optimized stratified sampling for approximate query processing. *ACM Transactions on Database Systems (TODS) 32*, 2 (2007), 9.

[11] CHAUDHURI, S., DAS, G., AND SRIVASTAVA, U. Effective use of block-level sampling in statistics estimation. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data* (2004), ACM, p. 298.

[12] CHROMY, J. Design optimization with multiple objectives. In *Proceedings on the Research Methods of the American Statistical Association* (1987), pp. 194–199.

[13] CNET. Security guide to customs-proofing your laptop. *http://news.cnet.com/8301-13578_3-9892897-38.html* (2009).

[14] COCHRAN, W. Sampling technique. *New York: John Willey & Sons* (1977).

[15] DAS, G. Survey of approximate query processing techniques (tutorial). In *International Conference on Scientific and Statistical Database Management (SSDBM '03)* (2003).

[16] DASGUPTA, A., DAS, G., AND MANNILA, H. A random walk approach to sampling hidden databases. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data (SIGMOD '07)* (2007), pp. 629–640.

[17] DASGUPTA, A., JIN, X., JEWELL, B., ZHANG, N., AND DAS, G. Unbiased estimation of size and other aggregates over hidden web databases. In *Proceedings of the 2010 international conference on Management of data (SIGMOD)* (2010), pp. 855–866.

[18] DASGUPTA, A., ZHANG, N., AND DAS, G. Leveraging count information in sampling hidden databases. In *Proceedings of the 2009 IEEE International Conference on Data Engineering* (2009), pp. 329–340.

[19] DASGUPTA, A., ZHANG, N., DAS, G., AND CHAUDHURI, S. Privacy preservation of aggregates in hidden databases: why and how? In *Proceedings of the 35th SIGMOD international conference on Management of data* (2009), pp. 153–164.

[20] DAVID, H. A., AND NAGARAJA, H. N. *Order Statistics (3rd Edition)*. Wiley, New Jersey, 2003.

[21] ELLARD, D., LEDLIE, J., MALKANI, P., AND SELTZER, M. Passive nfs tracing of email and research workloads. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03)* (Berkeley, CA, USA, 2003), USENIX Association, pp. 203–216.

[22] GAROFALAKIS, M. N., AND GIBBON, P. B. Approximate query processing: Taming the terabytes. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB)* (2001).

[23] GOOGLE. Google desktop. *http://desktop.google.com/*.

[24] HEDLEY, Y. L., YOUNAS, M., JAMES, A., AND SANDERSON, M. A two-phase sampling technique for information extraction from hidden web databases. In *Proceedings of the 6th annual ACM international workshop on Web information and data management (WIDM '04)* (2004), pp. 1–8.

[25] HEDLEY, Y.-L., YOUNAS, M., JAMES, A. E., AND SANDERSON, M. Sampling, information extraction and summarisation of hidden web databases. *Data and Knowledge Engineering 59*, 2 (2006), 213–230.

[26] HUA, Y., JIANG, H., ZHU, Y., FENG, D., AND TIAN, L. SmartStore: a new metadata organization paradigm with semantic-awareness for next-generation file systems. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)* (2009), ACM, pp. 1–12.

[27] ILYAS, I. F., BESKALES, G., AND SOLIMAN, M. A. A survey of top-*k* query processing techniques in relational database systems. *ACM Computing Surveys 40*, 4 (2008), 1–58.

[28] IPEIROTIS, P. G., AND GRAVANO, L. Distributed search over the hidden web: hierarchical database sampling and selection. In *Proceedings of the 28th international conference on Very Large Data Bases (VLDB '02)* (2002), pp. 394–405.

[29] KOGGE, P., BERGMAN, K., BORKAR, S., CAMPBELL, D., CARLSON, W., DALLY, W., DENNEAU, M., FRANZON, P., HARROD, W., HILL, K., ET AL. Exascale computing study: technology challenges in achieving exascale systems. *DARPA Information Processing Techniques Office 28* (2008).

[30] LEUNG, A. Organizing, indexing, and searching large-scale file systems. Tech. Rep. UCSC-SSRC-09-09, University of California, Santa Cruz, Dec. 2009.

[31] LEUNG, A., ADAMS, I., AND MILLER, E. Magellan: a searchable metadata architecture for large-scale file systems. Tech. Rep. UCSC-SSRC-09-07, University of California, Santa Cruz, Nov. 2009.

[32] LEUNG, A. W., SHAO, M., BISSON, T., PASUPATHY, S., AND MILLER, E. L. Spyglass: fast, scalable metadata search for large-scale storage systems. In *Proccedings of the 7th conference on File and Storage Technologies (FAST)* (Berkeley, CA, USA, 2009), USENIX Association, pp. 153–166.

[33] LILLIBRIDGE, M., ESHGHI, K., BHAGWAT, D., DEOLALIKAR, V., TREZISE, G., AND CAMBLE, P. Sparse indexing: large scale, inline deduplication using sampling and locality. In *Proccedings of the 7th conference on File and Storage Technologies (FAST)* (Berkeley, CA, USA, 2009), USENIX Association, pp. 111–123.

[34] LOHR, S. Sampling: design and analysis. *Pacific Grove* (1999).

[35] LYNN, W. J. Defending a new domain: the pentagon's cyberstrategy. *Foreign Affairs* (September/October 2010).

[36] MURPHY, N., TONKELOWITZ, M., AND VERNAL, M. The design and implementation of the database file system. *http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.11.8068*.

[37] NUNEZ, J. High end computing file system and IO R&D gaps roadmap. *HEC FSIO R&D Conference* (Aug. 2008).

[38] OLKEN, F., AND ROTEM, D. Simple random sampling from relational databases. In *Proceedings of the 12th International Conference on Very Large Data Bases* (1986), pp. 160–169.

[39] OLKEN, F., AND ROTEM, D. Random sampling from database files: a survey. In *Proceedings of the fifth international conference on Statistical and scientific database management* (1990), Springer-Verlag New York, Inc., pp. 92–111.

[40] OLSON, M. The design and implementation of the Inversion file system. In *Proceedings of the Winter 1993 USENIX Technical Conference* (1993), pp. 205–217.

[41] PIKE, R., PRESOTTO, D., DORWARD, S., FLANDRENA, B., THOMPSON, K., TRICKEY, H., AND WINTERBOTTOM, P. Plan 9 from bell labs. *Computing systems 8*, 3 (1995), 221–254.

[42] PLAN 9 FILE SYSTEM TRACES. *http://pdos.csail.mit.edu/p9trace/*.

[43] SELTZER, M., AND MURPHY, N. Hierarchical file systems are dead. In *Proceedings of the 12th conference on Hot topics in Operating Systems (HotOS '09)* (2009), pp. 1–1.

[44] SLASHDOT. Laptops can be searched at the border. *http://yro.slashdot.org/article.pl?sid=08/04/22/1733251* (2008).

[45] SNIA. NFS traces. *http://iotta.snia.org/traces/list/NFS* (2010).

[46] STAHLBERG, P., MIKLAU, G., AND LEVINE, B. N. Threats to privacy in the forensic analysis of database systems. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data (SIGMOD '07)* (New York, NY, USA, 2007), ACM, pp. 91–102.

[47] SZALAY, A. New challenges in petascale scientific databases. In *Proceedings of the 20th international conference on Scientific and Statistical Database Management (SSDBM '08)* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 1–1.

[48] VITTER, J. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS) 11*, 1 (1985), 57.

[49] ZHU, Y., JIANG, H., WANG, J., AND XIAN, F. HBA: Distributed Metadata Management for Large Cluster-Based Storage Systems. *IEEE Transactions on Parallel and Distributed Systems 19* (2008), 750–763.

Making the Common Case the Only Case with Anticipatory Memory Allocation

Swaminathan Sundararaman, Yupu Zhang, Sriram Subramanian,
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
Computer Sciences Department, University of Wisconsin–Madison
{*swami,yupu,srirams,dusseau,remzi*}*@cs.wisc.edu*

Abstract

We present Anticipatory Memory Allocation (AMA), a new method to build kernel code that is robust to memory-allocation failures. AMA avoids the usual difficulties in handling allocation failures through a novel combination of static and dynamic techniques. Specifically, a developer, with assistance from AMA static analysis tools, determines how much memory a particular call into a kernel subsystem will need, and then pre-allocates said amount immediately upon entry to the kernel; subsequent allocation requests are serviced from the pre-allocated pool and thus guaranteed never to fail. We describe the static and run-time components of AMA, and then present a thorough evaluation of Linux ext2-mfr, a case study in which we transform the Linux ext2 file system into a memory-failure robust version of itself. Experiments reveal that ext2-mfr avoids memory-allocation failures successfully while incurring little space or time overhead.

1 Introduction

A great deal of recent activity in systems research has focused on new techniques for finding bugs in large code bases [13, 16, 17, 20, 24, 26, 38]. Whether using static analysis [16, 20], model checking [25, 40], symbolic execution [10, 39], machine learning [24], or other testing-based techniques [3, 4, 31], all seem to agree: there are hundreds of bugs in commonly-used systems.

One important class of software defect is found in *recovery code*, i.e., code that is run in reaction to failure. These failures, whether from hardware (e.g., a disk) or software (e.g., a memory allocation), tend to occur quite rarely in practice, but the correctness of the recovery code is critical. For example, Yang et al. found a large number of bugs in file-system recovery code; when such bugs were triggered, the results were often catastrophic, resulting in data corruption or unmountable file systems [40]. Recovery code has the worst possible property: it is rarely run, but absolutely must work correctly.

Memory-allocation failure serves as an excellent and important example of the recovery-code phenomenon. Woven throughout a complex system such as Linux are memory allocations of various flavors (e.g., `kmalloc`, `kmem_cache_alloc`, etc.) in conjunction with small snippets of recovery code to handle those rare cases when a memory allocation fails. As previous work has shown [17, 28, 40], and as we further demonstrate in this paper (§2), this recovery code does not work very well, often crashing the system or worse when run.

Thus, in this paper, we take a different approach to solving the problem presented by memory-allocation failures. We follow one simple mantra: *the most robust recovery code is recovery code that never runs at all.*

Our approach is called *Anticipatory Memory Allocation (AMA).* The basic idea behind AMA is simple. First, using both a static analysis tool plus domain knowledge, the developer determines a conservative estimate of the total memory allocation demand of each call into the kernel subsystem of interest. Using this information, the developer then augments their code to pre-allocate the requisite amount of memory at run-time, immediately upon entry into the kernel subsystem. The AMA run-time then transparently redirects existing memory-allocation calls to use memory from the pre-allocated chunk. Thus, when a memory allocation takes place deep in the heart of the kernel subsystem, it is guaranteed never to fail.

With AMA, kernel code is written naturally, with memory allocations inserted wherever the developer needs them to be; however, with AMA, the developer need not be concerned with downstream memory-allocation failures and the scattered (and often buggy) recovery code that would otherwise be required. Further, by allocating memory in one large chunk upon entry, failure of the anticipatory pre-allocation is straightforward to handle; a uniform failure-handling policy (such as retry with exponential backoff) can trivially be implemented.

To demonstrate the benefits of AMA, we apply it to the Linux ext2 file system to build a memory-failure robust version of ext2 called *ext2-mfr*. File systems are one of the most critical components of the kernel, as they store persistent state, and bugs within the file system can lead to serious problems [40]; hence, they serve as an excellent case study for AMA (although much of AMA is generic and could be applied elsewhere in the kernel). Through experiment, we show that ext2-mfr is robust to memory-allocation failure, and runs without noticeable performance or space overheads; key to the reduction in space overheads are two novel optimizations we introduce, *cache peeking* and *page recycling*. Further, very little code change is required, thus demonstrating the ease of transforming a significant subsystem. Overall, we find that AMA achieves its goals, and thus altogether avoids of one important class of recovery bug commonly found in kernel code.

In our current prototype, the static analysis tool in AMA is semi-automated. AMA requires developer involvement at the last stage of the static analysis to compute the memory requirements for each call. More programming effort is required to fully automate the static analysis tool. Hence, in its current form, our AMA prototype serves as a feasibility study of applying static analysis techniques inside operating systems to avoid a class of recovery code.

The rest of this paper is structured as follows. We first present more background on Linux memory allocation (§2), including a further study of how Linux file systems react to memory failure. We then present the design and implementation of AMA (§3,§4,§5), and evaluate its robustness and performance (§6). Finally, we discuss related work (§7) and conclude (§8).

2 Background

Before delving into the depths of AMA, we provide some background on kernel memory allocation. We first describe the many different ways in which memory is explicitly allocated within the kernel. Then, through fault injection, we show that many problems still exist in handling memory-allocation failures. Our discussion revolves around the Linux kernel (with a focus on file systems), although in our belief the issues that arise here likely exist in other modern operating systems.

2.1 Linux Allocators

2.1.1 Memory Zones

At the lowest level of memory allocation within Linux is a buddy-based allocator of physical pages [7], with low-level routines such as `alloc_pages()` and `free_pages()` called to request and return pages, respectively. These functions serve as the basis for the allocators used for kernel data structures (described below), although they can be called directly if so desired.

2.1.2 Kernel Allocators

Most dynamic memory requests in the kernel use the Linux *slab allocator*, which is based on Bonwick's original slab allocator for Solaris [6] (a newer SLUB allocator provides the same interfaces but is internally simpler). One simply calls the generic memory allocation routines `kmalloc()` and `kfree()` to use these facilities.

		kmem			
		cache		mempool	alloc
	kmalloc	alloc	vmalloc	create	pages
btrfs	93	7	3	0	1
ext2	8	1	0	0	0
ext3	12	1	0	0	0
ext4	26	10	1	0	0
jfs	18	1	2	1	0
reiser	17	1	5	0	0
xfs	11	1	0	1	1

Table 1: **Usage of Different Allocators.** *The table shows the number of different memory allocators used within Linux file systems. Each column presents the number of times a particular routine is found in each file system.*

For objects that are particularly popular, specialized caches can be explicitly created. To create such a cache, one simply calls `kmem_cache_create()`, which (if successful) returns a reference to the newly-created object cache; subsequent calls to `kmem_cache_alloc()` are passed this reference and return memory for the specific object. Hundreds of these specialized allocation caches exist in a typical system (see `/proc/slabinfo`); a common usage for a file system, for example, is an inode cache.

Beyond these commonly-used routines, there are a few other ways to request memory in Linux. A *memory pool* interface allows one to reserve memory for use in emergency situations. Finally, the *virtual malloc* interface requests in-kernel pages that are virtually (but not necessarily physically) contiguous.

To demonstrate the diversity of allocator usage, we present a study of the popularity of these interfaces within a range of Linux file systems. We study file systems as they are an important and complex kernel subsystem, and one in which memory-allocation failure can lead to serious problems [40]. Table 1 presents our results. As one can see, although the generic interface `kmalloc()` is most popular, the other allocation routines are used as well. For kernel code to be robust, it must handle failures from all of these allocation routines.

2.2 Failure Modes

When calling into an allocator, flags determine the exact behavior of the allocator, particularly in response to failure. Of greatest import to us is the use of the `__GFP_NOFAIL` flag, which a developer can use when they know their code cannot handle an allocation failure; using the flag is the only way to guarantee that an allocator will either return successfully or not return at all (i.e., keep trying forever). However, this flag is rarely used. As lead Linux kernel developer Andrew Morton said [27]: "`__GFP_NOFAIL` should only be used when we have no

	Process State		File-System State	
	Error	Abort	Unusable	Inconsistent
btrfs$_0$	0	0	0	0
btrfs$_{10}$	0	14	15	0
btrfs$_{50}$	0	15	15	0
ext2$_0$	0	0	0	0
ext2$_{10}$	10	5	5	0
ext2$_{50}$	10	5	5	0
ext3$_0$	0	0	0	0
ext3$_{10}$	10	5	5	4
ext3$_{50}$	10	5	5	5
ext4$_0$	0	0	0	0
ext4$_{10}$	10	5	5	5
ext4$_{50}$	10	5	5	5
jfs$_0$	0	0	0	0
jfs$_{10}$	15	0	2	5
jfs$_{50}$	15	0	5	5
reiserfs$_0$	0	0	0	0
reiserfs$_{10}$	10	4	4	0
reiserfs$_{50}$	10	5	5	0
xfs$_0$	0	0	0	0
xfs$_{10}$	13	1	0	3
xfs$_{50}$	10	5	0	5

Table 2: **Fault Injection Results.** *The table shows the reaction of the Linux file systems to memory-allocation failures as the probability of a failure increases. We randomly inject faults into the three most-used allocation calls:* `kmalloc()`*,* `kmem_cache_alloc()`*, and* `__alloc_pages()`*. For each file system and each probability (shown as subscript), we run a micro benchmark 15 times and report the number of runs in which certain failures happen in each column. We categorize all failures into process state and file-system state, in which ’Error’ means that file system operations fail (gracefully), ’Abort’ indicates that the process was terminated abnormally, ’Unusable’ means the file system is no longer accessible, and ’Inconsistent’ means file system metadata has been corrupted and data may have been lost. Ideally, we expect the file systems to gracefully handle the error (i.e., return error) or retry the failed allocation request. Aborting a process, inconsistent file-system state, and unusable file system are unacceptable actions on an memory allocation failure.*

way of recovering from failure. ... Actually, nothing in the kernel should be using `__GFP_NOFAIL`. It is there as a marker which says ’we really shouldn’t be doing this but we don’t know how to fix it’.” In all other uses of kernel allocators, failure is thus a distinct possibility.

2.3 Bugs in Memory Allocation

Earlier work has repeatedly found that memory-allocation failure is often mishandled [16, 40]. In Yang et al.’s model-checking work, one key to finding bugs is to follow the code paths where memory-allocation has failed [40].

We now perform a brief study of memory-allocation failure handling within Linux file systems. We use fault injection to fail calls to the various memory allocators and determine how the code reacts as the number of such failures increases. Our injection framework picks a certain allocation call (e.g., `kmalloc()`) within the code and fails it probabilistically; we then vary the probability and observe how the kernel reacts as an increasing percentage of memory-allocation calls fail. Table 2 presents our results, which sums the failures seen in 15 runs per file system, while increasing the probability of an allocation request failing from 0% to 50% of the time.

```
empty_dir() [file: namei.c]
  if (...|| !(bh = ext4_bread(..., &err)))
    ...
    return 1; // XXX: should have returned 0


ext4_rmdir() [file: namei.c]
  retval = -ENOTEMPTY;
  if (!empty_dir(inode))
    goto end_rmdir;
  retval = ext4_delete_entry(handle, dir, de, bh);
  if (retval)
    goto end_rmdir;
```

Figure 1: **Improper Failure Propagation.** *The code shown in the figure is from the ext4 file system, and shows a case where a failed low-level allocation (in ext4_bread()) is not properly handled, which eventually leads to an inconsistent file system.*

The table reports what happens as the probability of allocation failure occurring increases, from 0% (base case), to 10% and then 50% of calls. We report the outcomes in two categories: process state and file-system state. The process state results are further divided into two groups: the number of times (in 15 runs) that a running process received an error (such as ENOMEM), and the number of times that a process was terminated abnormally (i.e., killed). The file system results are split into two categories as well: a count of the number of times that the file system became unusable (i.e., further use of the file system was not possible after the trial), and the number of times the file system became inconsistent as a result, possible losing user data.

From the table, we can make the following observations. First, we can see that even a simple, well-tested, and slowly-evolving file system such as Linux ext2 still does not handle memory-allocation failures very well; we take this as evidence that doing so is challenging. Second, we observe that all file systems have difficulty handling memory-allocation failure, often resulting in an unusable or inconsistent file system.

An example of how a file-system inconsistency can arise is found in Figure 1. In this example, in trying to remove a directory (in `ext4_rmdir()`), the routine first checks if the directory is empty by calling `empty_dir()`. This routine, in turn, calls `ext4_bread()` to read the directory data. Unfortunately, due to our fault injection, `ext4_bread()` tries to allocate memory but fails to do so, and thus the call to `ext4_bread()` returns an error (correctly). The routine `empty_dir()` incorrectly propagates this error, simply returning a 1 and thus accidentally indicating that the directory is empty and can be deleted. Deleting a non-empty directory not only leads to a hard-to-detect file-system inconsistency (despite the presence of journaling), but also could render inaccessible a large portion of the directory tree.

Finally, a closer look at the code of some of these file systems reveals a third interesting fact: in a file system under active development (such as btrfs), there are many places within the code where memory-allocation failure is never checked for; our inspection thus far has yielded over 20 places within btrfs such as this. Such trivial mishandling is rarer inside more mature file systems.

Overall, our results hint at a broader problem, which matches intuition: developers write code as if memory allocation will never fail; only later do they (possibly) go through the code and attempt to "harden" it to handle the types of failures that might arise. Proper handling of such errors, as seen in the ext4 example, is a formidable task, and as a result, such hardening sometimes remains "softer" than desired.

2.3.1 Summary

Kernel memory allocation is complex, and handling failures still proves challenging even for code that is relatively mature and generally stable. We believe these problems are fundamental given the way current systems are designed; specifically, to handle failure correctly, a *deep recovery* must take place, where far downstream in the call path, one must either handle the failure, or propagate the error up to the appropriate error-handling location while concurrently making sure to unwind all state changes that have taken place on the way down the path. Earlier work has shown that the simple act of propagating an error correctly in a complex file system is challenging [19]; doing so and correctly reverting all other state changes presents further challenges. Although deep recovery is possible, we believe it is usually quite hard, and thus error-prone. More sophisticated bug-finding tools could be built, and further bugs unveiled; however, to truly solve the problem, an alternate approach to deep recovery is likely required.

3 Anticipatory Memory Allocation: An Overview

We now present an overview of *Anticipatory Memory Allocation (AMA)*, a novel approach to solve the memory-allocation failure-handling problem. The basic idea is simple: first, we analyze the code paths of a kernel subsystem to determine what their memory requirements are. Second, we augment the code with a call to pre-allocate the necessary amounts. Third, we transparently redirect allocation requests during run-time to use the pre-allocated chunks of memory.

```
void f2() {
  void *p = malloc(100);
  f3();
}

void f3() {
  void *q = malloc(25);
}

int f1() {
  // AMA: Pre-allocate 100- and 25-byte chunks
  f2();
  // AMA: Free any unused chunks
}
```

Figure 2: **Simple AMA Example.** *The code presents a simple example of how AMA is used. In the unmodified case, routine `f1()` calls `f2()`, which calls `f3()`, each of which allocate some memory (and perhaps incorrectly handle their failure). With AMA, `f1()` pre-allocates the full amount needed; subsequent calls to allocate memory are transparently redirected to use the pre-allocated chunks instead of calling into the real allocators, and any remaining memory is freed.*

Figure 2 shows a simple example of the transformation. In the figure, a simple entry-point routine `f1()` calls one other *downstream* routine, `f2()`, which in turn calls `f3()`. Each of these routines allocates some memory during their normal execution, in this case 100 bytes by `f2()` and 25 bytes by `f3()`.

With AMA, we analyze the code paths to discover the worst-case allocation possible; in this example, the analysis would be simple, and the result is that two memory chunks, of size 100 and 25 bytes, are required. Then, before calling into `f2()`, one should call into the anticipatory memory allocator to pre-allocate chunks of 100 and 25 bytes. The modified run-time then redirects all downstream allocation requests to use this pre-allocated pool. Thus the calls to allocate 100 and 25 bytes in `f2()` and `f3()` (respectively) will use memory already allocated by AMA, and are guaranteed not to fail.

The advantages of this approach are many. First, memory-allocation failures never happen downstream, and thus there is no need to handle said failures; the complex unwinding of kernel state and error propagation are thus avoided entirely. Second, because allocation failure can only happen in only one place in the code (at the top), it is easy to provide a unified handling mechanism; for example, if the call to pre-allocate memory fails, the developer could decide to immediately return a failure, retry, or perhaps implement a more sophisticated exponential backoff-and-retry approach, all excellent examples of the *shallow recovery* AMA enables. Third, very little code change is required; except for the calls to pre-allocate and perhaps free unused memory, the bulk of the code remains

```
void
ext2_init_block_alloc_info(struct inode *inode)
{
  struct ext2_inode_info *ei = EXT2_I(inode);
  struct ext2_block_alloc_info *block_i =
        ei→i_block_alloc_info;
  block_i = kmalloc(sizeof(*block_i), GFP_NOFS);
  ...
}
```

Figure 3: **A Simple Call.**

unmodified, as the run-time transparently redirects downstream allocation requests to use the pre-allocated pool.

Unfortunately, code in real systems is not as simple as that found in the figure, and indeed, the problem of determining how much memory needs to be allocated given an entry point into a complex code base is generally undecidable. Thus, the bulk of our challenge is transforming the code and gaining certainty that we have done so correctly and efficiently. To gain a better understanding of the problem, we must choose a subsystem to focus upon, and transform it to use AMA.

3.1 A Case Study: Linux ext2-mfr

The case study we use is the Linux ext2 file system. Although simpler than its modern journaling cousins, ext2 is a real file system and certainly has enough complex memory-allocation behavior (as described below) to demonstrate the intricacies of developing AMA for a real kernel subsystem.

We describe our effort to transform the Linux ext2 file system into a memory-robust version of itself, which we call Linux ext2-mfr (i.e., a version of ext2 that is Memory-Failure Robust). In our current implementation, the transformation requires some human effort and is aided by a static analysis tool that we have developed. The process could be further automated, thus easing the development of other memory-robust file systems; we leave such efforts to future work.

We now highlight the various types of allocation requests that are made, from simpler to more complex. By doing so, we are showing what work needs to be done to be able to correctly pre-allocate memory before calling into ext2 routines, and thus shedding light on the types of difficulties we encountered during the transformation process.

3.1.1 Simple Calls

Most of the memory-allocation calls made by the kernel are of a fixed size. Allocating file system objects such as dentry, file, inode, page have pre-determined sizes. For example, file systems often maintain a cache of inode objects, and thus must have memory allocated for them before being read from disk. Figure 3 shows one example of such a call from ext2.

```
struct dentry *d_alloc(..., struct qstr *name) {
  ...
  if (name→len > DNAME_INLINE_LEN-1) {
    dname = kmalloc(name→len + 1, GFP_KERNEL);
    if (!dname)
      return NULL;
    ...
  }
}
```

Figure 4: **A Parameterized and Conditional Call.**

```
ext2_find_entry (struct inode * dir, ...)
{
  unsigned long npages = dir_pages(dir);
  unsigned long n = 0;
  do {
    page = ext2_get_page(dir, n,..); // allocate a page
    ...
    if (ext2_match_entry (...));
        goto found;
    ...
    n++;
  } while (n != npages); // worst case: n = npages
found:
 return entry;
}
```

Figure 5: **Loop Calls.**

3.1.2 Parameterized and Conditional Calls

Some allocated objects have variable lengths (e.g., a file name, extended attributes, and so forth) and the exact size of the of the allocation is determined at run-time; sometimes allocations are not performed due to conditionals. Figure 4 shows how ext2 allocates memory for a directory entry, which uses a length field (plus one for the end-of-string marker) to request the proper amount of memory. This allocation is only performed if the name is too long and requires more space to hold it.

3.1.3 Loops

In many cases file systems allocate objects inside a loop or inside nested loops. In ext2, the upper bound of the loop execution is determined by the object passed to the individual calls. For example, allocating pages to search for directory entries are done inside a loop. Another good example is searching for a free block within the block bitmaps of the file system. Figure 5 shows the page allocation code during directory lookups in ext2.

3.1.4 Function Calls

Of course, a file system is spread across many functions, and hence any attempt to understand the total memory allocation of a call graph given an entry point must be able to follow all such paths, sometimes into other major kernel subsystems. For example, one memory allocation request in ext2 is invoked 21 calls deep; this example path starts at `sys_open`, traverses through some link-traversal and lookup code, and ends with a call to `kmem_cache_alloc`.

```
static void
ext2_free_branches(struct inode *inode,
                         .., int depth){
  if (depth--) {
    ...
    // allocate a page and buffer head
    bh = sb_bread(inode→i_sb, ..);
    ...
    ext2_free_branches(inode,
                  (__le32*) bh→b_data,
                  (__le32*) bh→b_data +
                            addr_per_block,
                  depth);
  } else
    ext2_free_data(inode, ...);
}
```

Figure 6: **Recursion.**

3.1.5 Recursions

A final example of an in-kernel memory allocation is one that is performed within a recursive call. Some portions of file systems are naturally recursive (e.g., pathname traversal), and thus perhaps it is no surprise that recursion is commonplace. Figure 6 shows the block-freeing code that is called when a file is truncated or removed in ext2; in the example, `ext2_free_branches` calls itself to recurse down indirect-block chains and free blocks as need be.

3.2 Summary

To be able to pre-allocate enough memory for a call, one must handle parameterized calls, conditionals, loops, function calls, and recursion. If file systems only contained simple allocations and minimal amounts of code, pre-allocation would be rather straightforward. The relevant portion of the call graph for ext2 (and all related components of the kernel) contains nearly 2000 nodes (one per relevant function) and roughly 7000 edges (calls between functions) representing roughly 180,000 lines of kernel source code. Even for a relatively-simple file system such as ext2, the task of manually computing the pre-allocation amount would be daunting, without automated assistance.

4 The Static Transformation: From ext2 to ext2-mfr

We now present the static-analysis portion of AMA, in which we develop a tool, *the AMAlyzer*, to help decide how much memory to pre-allocate at each entry point into the kernel subsystem that is being transformed (in this case, Linux ext2). The AMAlyzer takes in the entire relevant call graph and produces a skeletal version, from which the developer can derive the proper pre-allocation amounts. After describing the tool, we also present two novel optimizations we employ, cache peeking and page recycling, to reduce memory demands. We end the section with a discussion of the limits of our current approach.

We build the AMAlyzer on top of CIL [29], a tool which allows us to readily analyze kernel source code. CIL does not resolve function pointers automatically, which we require for our complete call graph, and hence we perform a small amount of extra work to ensure we cover all calls made in the context of the file system; because of the limited and stylized use of function pointers within the kernel, this process is straightforward. The AMAlyzer in its current form is comprised of a few thousand lines of OCaml code.

4.1 The AMAlyzer

We now describe the AMAlyzer in more detail, which consists of two phases. In the first phase, the tool searches through the entire subsystem to construct the allocation-relevant call graph, i.e., the complete set of downstream functions that contain kernel memory-allocation requests. In the second phase, a more complex analysis determines which variables and state are relevant to allocation calls, and prunes away other irrelevant code. The result is a skeletal form of the subsystem in question, from which the pre-allocation amounts are readily derived.

4.1.1 Phase 1: Allocation-Relevant Call Graph

The first step of our analysis prunes the entire call graph, which, as we have seen, is quite large, and generates what we refer to as the *allocation-relevant call graph (ARCG)*. The ARCG contains only nodes and edges in which a memory allocation occurs, either within a node of the graph or somewhere downstream of it.

We perform a Depth First Search (DFS) on the call graph to generate ARCG. An additional attribute namely *calls_memory_allocation* is added to each node (i.e., function) in the call graph to speed up the ARCG generation. The calls_memory_allocation attribute is set on two occasions. First, when a memory allocation routine is encountered during the DFS. Second, the calls_memory_allocation attribute is set if at least one of the node's children has its calls_memory_allocation attribute set.

At the end of the DFS, the functions that do not have calls_memory_allocation attribute set are safely deleted from the call graph. The remaining nodes in the call graph constitute the ARCG.

4.1.2 Phase 2: Loops and Recursion

At this point, the tool has reduced the number of functions that must be examined. In this part of the analysis, we add logic to handle loops and recursions, and where possible, to help identify their termination conditions. The AMAlyzer searches for all `for`, `while`, and `goto`-based loops, and walks through each function within such a loop to find either direct calls to kernel memory allocators or indirect calls through other routines. To identify goto-based loops, AMA uses the line numbers of the labels that the goto statements point to. To identify both recursions

Entry point	Pre-allocation required
truncate()	$(Worst(Bitmap) + Worst(Indirect)) \times (PageSize + BufferHead)$
lookup()	$(1 + Size(ParentDir)) \times (PageSize + BufferHead) + Inode + Dentry + NameLength + NamesCache$
lookuphash()	$(1 + Size(ParentDir)) \times (PageSize + BufferHead) + Inode + Dentry + NameLength + Filp$
sysopen()	$lookup() + lookuphash() + (4 + Depth(Inode) + Worst(Bitmap)) \times PageSize + (5 + Depth(Inode) + Worst(Bitmap)) \times BufferHead + Inode + truncate()$
sysread()	$(count + ReadAhead + Worst(Bitmap) + Worst(Indirect)) \times (PageSize + BufferHead)$
syswrite()	$(count + Worst(Bitmap)) \times (PageSize + BufferHead) + sizeof(ext2\_block\_allocinfo)$
mkdir()	$lookup() + lookuphash() + (Depth(ParentInode) + 4) \times PageSize + (Depth(Inode) + 8) \times BufferHead$
unlink()	$lookup() + lookuphash() + (1 + Depth(Inode)) \times (PageSize + BufferHead)$
rmdir()	$lookup() + lookuphash() + (3 + Depth(Inode)) \times (PageSize + BufferHead)$
access()	$lookup() + NamesCache$
chdir()	$lookup() + NamesCache$
chroot()	$lookup() + NamesCache$
statfs()	$lookup() + NamesCache$

Table 3: **Pre-Allocation Requirements for ext2-mfr.** *The table shows the worst-case memory requirements of the various system calls in terms of the kmem_cache, kmalloc, and page allocations. The following types of kmem_cache are used: $NamesCache$ (4096 bytes), $BufferHead$ (52 bytes), $Inode$ (476 bytes), $Filp$ (128 bytes), and $Dentry$ (132 bytes). The $PageSize$ is constant at 4096 bytes. The other terms used above include: $Count$: the number of blocks read/written, $ReadAhead$: the number of read-ahead blocks, $Worst(Bitmap)$: the number of bitmap blocks that needs to be read, $Worst(Indirect)$: the number of indirect blocks to be read for that particular block, $Depth(inode)$: the maximum number of indirect blocks to be read for that particular inode, and $Size(inode)$: the number of pages in the inode.*

and function-call based loops, AMA performs a DFS on the ARCG and for every function encountered during the search, it checks if the function has been explored before. Once these loops are identified, the tool searches for and outputs the expressions that affect termination.

4.1.3 Phase 3: Slicing and Backtracking

The goal of this next step is to perform a bottom-up crawl of the graph, and produce a minimized call graph with only the memory-relevant code left therein. We use a form of backward slicing [37] to achieve this end.

In our current prototype, the AMAlyzer only performs a bottom-up crawl until the beginning of each function. In other words, the slicing is done at the function level and developer involvement is required to perform backtracking. To backtrack until the beginning of a system call, the developer has to manually use the output of slicing for each function (including the dependent input variables that affect the allocation size/count) and invoke the slicing routine on its caller functions. The caller functions are identified using the ARCG.

4.2 AMAlyzer Summary

As we mentioned above, the final output is a skeletal graph which can be used by the developer to arrive at the final pre-allocations with the help of slicing support in the AMAlyzer. For ext2-mfr, the reduction in code is dramatic: from nearly 200,000 lines of code across 2000 functions (7000 function calls) down to less than 9,000 lines across 300 functions (400 function calls), with all relevant variables highlighted. Arriving upon the final pre-allocation amounts then becomes a straightforward process.

Table 3 summarizes the results of our efforts. In the table, we present the parameterized memory amounts that must be pre-allocated for the 13 most-relevant entry points into the file system.

4.3 Optimizations

As we transformed ext2 into ext2-mfr, we noticed a number of opportunities for optimization, in which we could reduce the amount of memory pre-allocated along some paths. We now describe two novel optimizations.

4.3.1 Cache Peeking

The first optimization, *cache peeking*, can greatly reduce the amount of pre-allocated memory. An example is found in code paths that access a file block (such as a `sys_read()`). To access a file block in a large file, it is possible that a triple-indirect, double-indirect, and indirect block, inode, and other blocks may need to be accessed to find the address of the desired block and read it from disk.

With repeated access to a file, such blocks are likely to be in the page cache. However, the pre-allocation code must account for the worst case, and thus in the normal case must pre-allocate memory to potentially read those blocks. This pre-allocation is often a waste, as the blocks will be allocated, remain unused during the call, and then finally be freed by AMA.

With cache peeking, the pre-allocation code performs a small amount of extra work to determine if the requisite pages are already in cache. If so, it pins them there and avoids the pre-allocation altogether; upon completion, the pages are unpinned.

The pin/unpin is required for this optimization to be safe. Without this step, it would be possible that a page gets evicted from the cache after the pre-allocation phase but before the use of the page, which would lead to an unexpected memory allocation request downstream. In this case, if the request then failed, AMA would not have served its function in ensuring that no downstream failures occur.

Cache peeking works well in many instances as the cached data is accessible at the beginning of a system call and does not require any new memory allocations. Even if cache peeking requires additional memory, the memory allocation calls needed for cache peeking can be easily performed as part of the pre-allocation phase.

4.3.2 Page Recycling

A second optimization we came upon was the notion of *page recycling*. The idea for the optimization arose when we discovered that ext2 often uses far more pages than needed for certain tasks (such as file/directory truncates, searches on free/allocated entries inside block bitmaps and large directories).

For example, consider truncate. In order to truncate a file, one must read every indirect block (and double indirect block, and so forth) into memory to know which blocks to free. In ext2, each indirect block is read into memory and given its own page; the page holding an indirect block is quickly discarded, after ext2 has freed the blocks pointed to by that indirect block.

To reduce this cost, we implement page recycling. With this approach, the pre-allocation phase allocates the minimal number of pages that need to be in memory during the operation. For a truncate, this number is proportional to the depth of the indirect-block tree, instead of the size of the entire tree. Instead of allocating thousands of blocks to truncate a file, we only allocate a few (for the triple-indirect, a double indirect, and an indirect block). When the code has finished freeing the current indirect block, we recycle that page for the next indirect block instead of adding the page back to the LRU page cache, and so forth. In this manner, substantial savings in memory is made possible.

4.4 Limitations and Discussion

We now discuss some of the limitations of our anticipatory approach.

Not all pieces are yet automated; instead, the tool currently helps turn the intractable problem of examining 180,000 lines of code into a tractable one providing a lot of assistance in finding the correct pre-allocations. Further work is required in slicing and backtracking to streamline this process, but is not the focus of our current effort: rather our goal here is to demonstrate the feasibility of the anticipatory approach.

The anticipatory approach could fail requests in cases where normal execution would successfully complete. Normal execution need not always take the worst case (or longest) path. As a result, it might be able to complete with fewer memory allocations than the anticipatory approach. In contrast, anticipatory approach has to always allocate memory for the worst case scenario, as it cannot afford to fail on a memory allocation call after the pre-allocation phase.

Cache peeking can only be used when sufficient information is available at the time of allocation to determine if the required data is in the cache. Sufficient information is available for file systems at the beginning of a system call in the context of file/directory reads and lookup of file-system objects, this allows cache peeking to avoid pre-allocation with little implementation effort. More implementation effort could be required in other systems to help determine if the required data is in its cache.

5 The AMA Run-Time

The final piece of AMA is the runtime component. There are two major pieces to consider. First is the pre-allocation itself, which is inserted at every relevant entry point in the kernel subsystem of interest, and subsequent cleanup of pre-allocated memory. Second is the use of the pre-allocated memory, in which the run-time must transparently redirect allocation requests (such as `kmalloc()`) to use the pre-allocated memory. We discuss these in turn, and then present the other run-time decision a file system such as Linux ext2-mfr must make: what to do when a pre-allocation request fails?

5.1 Pre-allocating and Freeing Memory

To add pre-allocation to a specific file system, we require that the file system to implement a single new VFS-level call, which we call `vfs_get_mem_requirements()`. This call takes as arguments information about which call is about to be made, any relevant arguments about the current operation (such as the file position or bytes to be read) and state of the file system, and then returns a structure to the caller (in this case, the VFS layer) which describes all of the necessary allocations that must take place. The structure is referred to as the *anticipatory allocation description (AAD)*.

The VFS layer unpacks the AAD, allocates memory chunks (perhaps using different allocators) as need be, and links them into the task structure of the calling process for downstream use (described further below). With the pre-allocated memory in place, the VFS layer then calls the desired routine (such as `vfs_read()`), which then

```
 loff_t pos = file_pos_read(file);
AMA_CHECK_AND_ALLOCATE(file,
                AMA_SYS_READ, pos, count);
 ret = vfs_read(file, buf, count, &pos);
 file_pos_write(file, pos);
AMA_CLEANUP();
```

Figure 7: **A VFS Read Example.**

utilizes the pre-allocated memory during its execution. When the operation completes, a generic AMA cleanup routine is called to free any unused memory.

To give a better sense of this code flow, we provide a simplified example from the `read()` system call code path in Figure 7. Without the AMA additions, the code simply looks up the current file position (i.e., where to read from next), calls into `vfs_read()` to do the file-system-specific read, updates the file offset, and returns. As described in the original VFS paper [23], this code is generic across all file systems.

With AMA, two extra steps are required, as shown in the figure. First, before calling into the `vfs_read()` call, the VFS layer now checks if the underlying file system is using AMA, and if so, calls the file system's `vfs_get_mem_requirements()` routine to determine the pending call's memory requirements, and finally allocates the needed memory. All of this work is neatly encapsulated by the `AMA_CHECK_AND_ALLOCATE()` call in the figure.

Second, after the call is complete, a cleanup routine `AMA_CLEANUP()` is called. This call is required because the AMAlyzer provides us with a worst-case estimate of possible memory usage, and hence not all pre-allocated memory is used during the course of a typical call into the file system. In order to free this unused memory, the extra call to `AMA_CLEANUP()` is made.

5.2 Using Pre-allocated Memory

Central to our implementation is *transparency*; we do not change the specific file system (ext2) or other kernel code to explicitly use or free pre-allocated memory. File systems and the rest of the kernel thus continue to use regular memory-allocation routines.

To support this transparency, we modified each of the kernel allocation routines as follows. Specifically, when a process calls into ext2-mfr, the pre-allocation code (in `AMA_CHECK_AND_ALLOCATE()` above) sets a new flag within the per-task task structure. This *anticipatory flag* is then checked upon each entry into any kernel memory-allocation routine. If the flag is set, the routine attempts to use pre-allocated memory and if so completes by returning one of the pre-allocated chunks; if the flag is not set, the normal allocation code is executed (and failure is a possibility). Calls to `kfree()` and other memory-releasing routines operate as normal, and thus we leave those unchanged.

Allocation requests are matched with the pre-allocated objects using the parameters passed to the allocation call at runtime. The parameters passed to the allocation call are size, order or the cachep pointer and the GFP flag. The type of the desired memory object is inferred through the invocation of the allocation call at runtime. The size (for kmalloc and vmalloc) or order (for alloc_pages) helps to exactly match the allocation request with the pre-allocated object. For cache objects, the cachep pointer help identify the correct pre-allocated object.

One small complication arises during interrupt handling. Specifically, we do not wish to redirect memory allocation requests to use pre-allocated memory when requested by interrupt-handling code. Thus, when interrupted, we take care to save the anticipatory flag of the currently-running process and restore it when the interrupt handling is complete.

5.3 What If Pre-Allocation Fails?

Adding the pre-allocation into the code raises a new policy question: how should the code handle the failure of the pre-allocation itself? We believe there are a number of different policy alternatives, which we now describe:

- **Fail-immediate.** This policy immediately returns an error to the caller (such as ENOMEM).
- **Retry-forever (with back-off).** This policy simply keeps retrying forever, perhaps inserting a delay of some kind (e.g., exponential) between retry requests to reduce the load on the system and control better the load on the memory system.
- **Retry-alternate (with back-off).** This form of retry also requests memory again, but uses an alternate code path that uses less memory than the original through page/memory recycling and thus is more likely to succeed. This retry can also back-off as need be.

Using AMA to implement these policies is superior to the existing approach, as it enables *shallow recovery*, immediately upon entry into the subsystem. For example, consider the fail-immediate option above. Clearly this policy *could* be implemented in the traditional system without AMA, but in our opinion doing so is prohibitively complex. To do so, one would have to ensure that the failure was propagated correctly all the way through the many layers of the file system code, which is difficult [19, 34]. Further, any locks acquired or other state changes made would have to be undone. Deep recovery is difficult and error-prone; shallow recovery is the opposite.

Another benefit that the shallow recovery of AMA permits is a unified policy. The policy, whether failing immediately, retrying, or some combination, is specified in one

	Process State		File-System State	
	Error	Abort	Unusable	Inconsistent
ext2-mfr$_{10}$	0	0	0	0
ext2-mfr$_{50}$	0	0	0	0
ext2-mfr$_{99}$	0	0	0	0

Table 4: **Fault Injection Results: Retry.** *The table shows the reaction of the Linux ext2-mfr file system to memory failures as the probability of a failure increases. The file system uses a "retry-forever" policy to handle each failure. A detailed description of the experiment is found in Table 2.*

or a few places in the code. Thus, the developer can easily decide how the system should handle such a failure and be confident that the implementation meets that desire.

A third benefit of our approach: file systems could expose some control over the policy to applications. Whereas most applications may not be prepared to handle such a failure, a more savvy application (such as a file server or database) could set the file system to fail-fast and thus enable better control over failure handling.

Pre-allocation failure is not a panacea, however. Depending on the installation and environment, the code that handles pre-allocation failures will possibly run quite rarely, and thus may not be as robust as normal-case code. Although we believe this to be less of a concern for pre-allocation recovery code (because it is small, simple, and usually correct "by inspection"), further efforts could be applied to harden this code. For example, some have suggested constant "fire drilling" [9] as a way to ensure operators are prepared to handle failures; similarly, one could regularly fail kernel subsystems (such as memory allocators) to ensure that this recovery code is run.

6 Analysis

We now analyze Linux ext2-mfr. We measure its robustness under memory-allocation failure, as well as its baseline performance. We further study its space overheads, exploring cases where our estimates of memory-allocation needs could be overly conservative, and whether the optimizations introduced earlier are effective in reducing these overheads. All experiments were performed on a 2.2 GHz Opteron processor, with two 80GB WDC disks, 2GB of memory, running Linux 2.6.32. We also experimented with the ramfs file system and were able to get similar performance results and better space overheads (not shown in the evaluation results).

6.1 Robustness

Our first experiment with ext2-mfr reprises our earlier fault injection study found in Table 2. In this experiment, we vary the probability that the memory-allocation routines will fail from 10% all the way to 99%, and observe how ext2-mfr behaves both in terms of how processes were affected as well as the overall file-system state. For this experiment, the retry-forever (without any back-off) policy is used. Table 4 reports our results.

As one can see from the table, ext2-mfr is highly robust to memory allocation failure. Even when 99 out of 100 memory-allocation calls fail, ext2-mfr is able to retry and eventually make progress. No application notices that the failures are occurring, and file system usability and state remain intact.

Workload	ext2 (secs)	ext2-mfr (secs)
Sequential Write	13.46	13.69 (1.02x)
Sequential Read	9.04	9.05 (1.01x)
Random Writes	11.58	11.67 (1.01x)
Random Reads	146.33	151.03 (1.03x)
Sort	129.64	136.50 (1.05x)
OpenSSH	48.30	49.80 (1.03x)
PostMark	55.90	59.60 (1.07x)

Table 5: **Baseline Performance.** *The baseline performance of ext2 and ext2-mfr are compared. The first four tests are microbenchmarks: sequential read and write either read or write 1-GB file in its entirety; random read and write read or write 100 MB of data over a 1-GB file. Note that random-write performance is good because the writes are buffered and thus can be scheduled when written to disk. The three application-level benchmarks: are a command-line sort of a 100MB text file; the OpenSSH benchmark which copies, untars, configures, and builds the OpenSSH 4.5.1 source code; and the PostMark benchmark run for 60,000 transactions over 3000 files (from 4KB to 4MB) with 50/50 read/append and create/delete biases. All times are reported in seconds, and are stable across repeated runs.*

6.2 Performance

In our next experiment, we study the performance overheads of using AMA. We utilize both simple microbenchmarks as well as application-level tests to gauge the overheads incurred in ext2-mfr due to the extra work of memory pre-allocation and cleanup. Table 5 presents the results of our study.

From the table, we can see that the performance of our relatively-untuned prototype is excellent across both microbenchmarks as well as application-level workloads. In all cases, the extra work done by the AMA runtime to pre-allocate memory, redirect allocation requests transparently, and subsequently free unused memory has a minimal cost. With further streamlining, we feel confident that the overheads could be reduced even further.

6.3 Space Overheads and Cache Peeking

We now study the space overheads of ext2-mfr, both with and without our cache-peeking optimization. The largest concern we have about conservative pre-allocation is that excess memory may be allocated and then freed; although we have shown there is little time overhead involved (Ta-

Workload	ext2 (GB)	ext2-mfr (GB)	ext2-mfr (+peek) (GB)
Sequential Read	1.00	6.89 (6.87x)	1.00 (1.00x)
Sequential Write	1.01	1.01 (1.00x)	1.01 (1.00x)
Random Read	0.26	0.63 (2.41x)	0.28 (1.08x)
Random Write	0.10	0.10 (1.05x)	0.10 (1.00x)
PostMark	3.15	5.88 (1.87x)	3.28 (1.04x)
Sort	0.10	0.10 (1.00x)	0.10 (1.00x)
OpenSSH	0.02	1.56 (63.29x)	0.07 (3.50x)

Table 6: **Space Overheads.** *The total amount of memory allocated for both ext2 and ext2-mfr is shown. The workloads are identical to those described in the caption of Table 5.*

	Process State		File-System State	
	Error	Abort	Unusable	Inconsistent
ext2-mfr$_{10}$	15	0	0	0
ext2-mfr$_{50}$	15	0	0	0
ext2-mfr$_{99}$	15	0	0	0

Table 7: **Fault Injection Results: Fail-Fast.** *The table shows the reaction of Linux ext2-mfr using a fail-fast policy file system. A detailed description of the experiment is found in Table 2.*

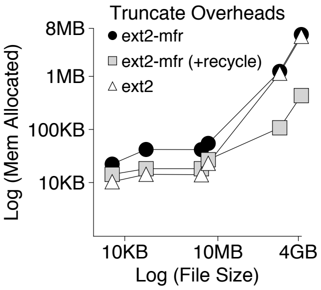


Figure 8: **Space Costs with Page Recycling.** *The figure shows the measured space overheads of page recycling during the truncate of a file. The file size is varied along the x-axis, and the space cost is plotted on the y-axis (both are log scales).*

ble 5), the extra space requested could induce further memory pressure on the system, (ironically) making allocation failure more likely to occur. We run the same set of microbenchmarks and application-level workloads, and record information about how much memory was allocated for both ext2 and ext2-mfr; we also turn on and off cache-peeking for ext2-mfr. Table 6 presents our results.

From the table, we make a number of observations. First, our unoptimized ext2-mfr does indeed conservatively pre-allocate a noticeable amount more memory than needed in some cases. For example, during a sequential read of a 1 GB file, normal ext2 allocates roughly 1 GB (mostly to hold the data pages), whereas unoptimized ext2-mfr allocates nearly seven times that amount. The file is being read one 4-KB block at a time, which means on average, the normal scan allocates one block per read whereas ext2-mfr allocates seven. The reason for these excess pre-allocations is simple: when reading a block from a large file, it is *possible* that one would have to read in a double-indirect block, indirect block, and so forth. However, as those blocks are already in cache for these reads, the conservative pre-allocation performs a great deal of unnecessary work, allocating space for these blocks and then freeing them immediately after each read completes; the excess pages are not needed.

With cache peeking enabled, the pre-allocation space overheads improve significantly, as virtually all blocks that are in cache need not be allocated. Cache peeking clearly makes the pre-allocation quite space-effective. The only workload which do not approach the minimum is OpenSSH. OpenSSH, however, places small demand on the memory system in general and hence is not of great concern.

6.4 Page Recycling

We also study the benefits of page recycling. In this experiment, we investigate the memory overheads of that arise during truncate. Figure 8 plots the results.

In the figure, we compare the space overheads of standard ext2, ext2-mfr (without cache peeking), and ext2-mfr with page recycling. As one can see from the figure, as the file system grows, the space overheads of both ext2 and ext2-mfr converge, as numerous pages are allocated for indirect blocks. Page recycling obviates the need for these blocks, and thus uses many fewer pages than even standard ext2.

6.5 Conservative Pre-allocation

We also were interested in whether, despite our best efforts, ext2-mfr ever under-allocated memory in the pre-allocation phase. Thus, we ran our same set of workloads and checked for this case. In no run during these experiments and other stress-tests did we ever encounter an under-allocation, giving us further confidence that our static transformation of ext2 was properly done.

6.6 Policy Alternatives

We also were interested in seeing how hard it is to use a different policy to react to allocation failures. Table 7 shows the results of our fault-injection experiment, but this time with a "fail-fast" policy which immediately returns to the user should the pre-allocation attempt fail.

The results show the expected outcome. In this case, the process running the workload immediately returns the `ENOMEM` error code; the file system remains consistent and usable. By changing only a few lines of code, an entirely different failure-handling behavior can be realized.

7 Related Work

A large body of related work is found in the programming languages community on heap usage analysis, wherein researchers have developed static analyses to determine how much heap (or stack) space a program will use [1, 8, 11, 12, 21, 22, 35, 36]. The general use-case suggested for said analyses is in the embedded domain, where memory and time resources are generally quite constrained [11]. Whereas many of the analyses focus on functional or garbage-collected languages, and thus are not directly applicable to our problem domain, we do believe that some of the more recent work in this space could be applicable to anticipatory memory allocation. In particular, Chin et al.'s work on analyzing "low-level" code [11] and the live heap analysis implemented by Albert et al. [1] are promising candidates for further automating the AMA transformation process.

The more general problem of handling "memory bugs" has also been investigated in great detail [2, 5, 14, 32, 33]; see Berger and Zorn for an excellent discussion of the range of common problems, including dangling pointers, double frees, and buffer overruns [5]. Many interesting and novel solutions have been proposed, including rolling back and trying again with a small change to the environment (e.g., more padding) [32], using multiple randomized heaps and voting to determine correctness [5], and even returning "made up" values when out-of-bounds memory is accessed [33]. The problem we tackle is both narrower and broader at once: narrower in that one could view the poor handling of an allocation failure as just one class of memory bug; broader in that true recovery from such a failure in a complex code base is quite intricate and reaches beyond the scope of typical solutions to these classic memory bugs.

Our approach of using static analysis to predict memory-requirement is similar in spirit to that taken by Garbervetsky et al. [18]. Their approach helps to come up with estimates of memory allocation within a given region. Moreover, their system does not consider the allocations done by native methods or internal allocation performed by the runtime system, and do not handle recursive calls. In contrast, AMA comes with the estimate for the entire file-system operation. Also, AMA estimates the allocations done by the kernel along with handling recursive calls inside file systems.

Our approach to avoiding memory-allocation failure is reminiscent of the banker's algorithm [15] and other deadlock-avoidance techniques. Indeed, with AMA, one could build a sort of "memory scheduler" that avoided memory over-commitment by delaying some requests until others frees had taken place, another avenue we plan to explore in future work.

Finally, our approach draws on concurrency control in its resemblance to two-phase locking [30], in which all locks are first acquired in an "expanding phase", then used, and then all released during a "shrinking phase". The expanding phase thus bears likeness to our pre-allocation request, in that all necessary resources are acquired up front before they are needed.

8 Conclusions

"Act as if it were impossible to fail." (Dorothea Brande)

It is common sense in the world of programming that code that is rarely run rarely works. Unfortunately, some of the most important code in systems falls into this category, including any code that is run during a "recovery". If the problem that leads to the recovery code being enacted is rare enough, the recovery code itself is unlikely to be battle tested, and is thus prone to failure.

We have presented Anticipatory Memory Allocation (AMA), a new approach to avoiding memory-allocation failures deep within the kernel. By pre-allocating the worst-case allocation immediately upon entry into the kernel, AMA ensures that requests further downstream will never fail, in those places within the code where handling failure has proven difficult over the years. The small bits of recovery code that are scattered throughout the code need never run, and system robustness is improved by design.

As we build increasingly complex systems, perhaps we should consider new methods and approaches that help build robustness into the system by design. AMA presents one method (early resource allocation) to handle one problem (memory-allocation failure), but we believe that the approach could be applied more generally. Our long term goal is to unify mainline code and recovery code into one; put another way, the only true manner in which to have working recovery code is to have none at all.

9 Acknowledgments

We thank the anonymous reviewers and Wilson Hsieh (our shepherd) for their feedback and comments, which have substantially improved the content and presentation of this paper. We also thank Joe Meehean and Laxman Visampalli for their comments on earlier drafts of the paper.

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] Elvira Albert, Samir Genaim, and Miguel Gomez-Zamalloa. Live Heap Space Analysis for Languages for Garbage Collection. In *International Symposium on Memory Management (ISMM '09)*, Dublin, Ireland, June 2009.

[2] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient Detection of All Pointer and Array Access Errors. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI '04)*, pages 290–301, Washington, DC, June 2004.

[3] Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Dependability Analysis of Virtual Memory Systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '06)*, Philadelphia, Pennsylvania, June 2006.

[4] Lakshmi N. Bairavasundaram, Meenali Rungta, Nitin Agrawal, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Systematically Benchmarking the Effects of Disk Pointer Corruption. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '08)*, Anchorage, Alaska, June 2008.

[5] Emery D. Berger and Benjamin G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI '06)*, Ottawa, Canada, June 2006.

[6] Jeff Bonwick. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *Proceedings of the USENIX Summer Technical Conference (USENIX Summer '94)*, Boston, Massachusetts, June 1994.

[7] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel.* O'Reilly, 2006.

[8] V. Braberman, F. Fernandez, D. Garbervetsky, and S. Yovine. Parametric Prediction of Heap Memory Requirements. In *International Symposium on Memory Management (ISMM '08)*, Tucson, Arizona, June 2008.

[9] Aaron B. Brown and David A. Patterson. To Err is Human. In *EASY '01*, 2001.

[10] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS '06)*, Alexandria, Virginia, November 2006.

[11] Wei-Ngan Chin, Huu Hai Nguyen, Corneliu Popeea, and Shengchao Qin. Analysing Memory Resource Bounds for Low-Level Programs. In *International Symposium on Memory Management (ISMM '08)*, Tucson, Arizona, June 2008.

[12] Wei-Ngan Chin, Huu Hai Nguyen, Shengchao Qin, and Martin Rinard. Memory Usage Verification for OO Programs. In *Static Analysis Symposium (SAS '05)*, 2005.

[13] 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.

[14] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory Safety Without Runtime Checks Or Garbage Collection. In *LCTES '03*, 2003.

[15] E. W. Dijkstra. EWD623: The Mathematics Behind The Bankers Algorithm. Selected Writings on Computing: A Personal Perspective (Springer-Verlag), 1977.

[16] 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.

[17] Dawson Engler and Madanlal Musuvathi. Static Analysis versus Software Model Checking for Bug Finding. In *5th International Conference Verification, Model Checking and Abstract Interpretation (VMCAI '04)*, Venice, Italy, January 2004.

[18] Diego Garbervetsky, Sergio Yovine, Víctor Braberman, Martín Rouaux, and Alejandro Taboada. On transforming java-like programs into memory-predictable code. In *JTRES '09: Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 140–149, New York, NY, USA, 2009. ACM.

[19] 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.

[20] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson R. Engler. A System and Language for Building System-Specific, Static Analyses. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '02)*, Berlin, Germany, June 2002.

[21] M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First Order Functional Languages. In *The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03)*, New Orleans, Louisiana, January 2003.

[22] Martin Hofmann and Steffen Jost. Type-based amortised heap-space analysis. In *In ESOP 2006, LNCS 3924*, pages 22–37. Springer, 2006.

[23] 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.

[24] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, California, December 2004.

[25] David Lie, Andy Chou, Dawson Engler, and David L. Dill. A Simple Method for Extracting Models from Protocol Code. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA '01)*, Goteborg, Sweden, June 2001.

[26] 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.

[27] Andrew Morton. Re: [patch] jbd slab cleanups. kerneltrap.org/mailarchive/linux-fsdevel/2007/9/19/322280/thread#mid-322280, September 2007.

[28] Madanlal Musuvathi, David Y.W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[29] George C. Necula, Scott McPeak, S. P. Rahul, and Westley Weimer. Cil: An infrastructure for c program analysis and transformation. In *International Conference on Compiler Construction (CC '02)*, pages 213–228, April 2002.

[30] Nathan Goodman Philip A. Bernstein, Vassos Hadzilacos. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[31] 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.

[32] 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.

[33] Martin Rinard, Christian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and Jr. William S. Beebe. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, California, December 2004.

[34] Cindy Rubio-Gonzalez, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. Error Propagation Analysis for File Systems. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation (PLDI '09)*, Dublin, Ireland, June 2009.

[35] TUGS. StackAnalyzer Stack Usage Analysis. http://www.absint.com/stackanalyzer/, September 2010.

[36] Leena Unnikrishnan and Scott D. Stoller. Parametric Heap Usage Analysis for Functional Programs. In *International Symposium on Memory Management (ISMM '09)*, Dublin, Ireland, June 2009.

[37] Mark Weiser. Program Slicing. In *International Conference on Software Engineering (ICSE '81)*, pages 439–449, San Diego, California, May 1981.

[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, Can Sar, Paul Twohey, Cristian Cadar, and Dawson Engler. Automatically Generating Malicious Disks using Symbolic Execution. In *IEEE Security and Privacy (SP '06)*, Berkeley, California, May 2006.

[40] 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.

Exploiting Memory Device Wear-Out Dynamics to Improve NAND Flash Memory System Performance

Yangyang Pan, Guiqiang Dong, and Tong Zhang
Electrical, Computer and Systems Engineering Department
Rensselaer Polytechnic Institute, USA.

Abstract

This paper advocates a device-aware design strategy to improve various NAND flash memory system performance metrics. It is well known that NAND flash memory program/erase (PE) cycling gradually degrades memory device raw storage reliability, and sufficiently strong error correction codes (ECC) must be used to ensure the PE cycling endurance. Hence, memory manufacturers must fabricate enough number of redundant memory cells geared to the worst-case device reliability at the end of memory lifetime. Given the memory device wear-out dynamics, the existing worst-case oriented ECC redundancy is largely *under-utilized* over the entire memory lifetime, which can be adaptively traded for improving certain NAND flash memory system performance metrics. This paper explores such device-aware adaptive system design space from two perspectives, including (1) how to improve memory program speed, and (2) how to improve memory defect tolerance and hence enable aggressive fabrication technology scaling. To enable quantitative evaluation, we for the first time develop a NAND flash memory device model to capture the effects of PE cycling from the system level. We carry out simulations using the DiskSim-based SSD simulator and a variety of traces, and the results demonstrate up to 32% SSD average response time reduction. We further demonstrate that the potential on achieving very good defect tolerance, and finally show that these two design approaches can be readily combined together to noticeably improve SSD average response time even in the presence of high memory defect rates.

1 Introduction

The steady bit cost reduction over the past decade has enabled NAND flash memory to enter increasingly diverse applications, from consumer electronics to personal and enterprise computing. In particular, it is now economically viable to implement solid-state drives (SSDs) using NAND flash memory, which is expected to fundamentally change the memory and storage hierarchy in future computing systems. As the semiconductor industry is aggressively pushing NAND flash memory technology scaling and the use of multi-bit-per-cell storage scheme, NAND flash memory increasingly relies on error correction codes (ECC) to ensure the data storage integrity. It is well known that NAND flash memory cells gradually wear out with the program/erase (PE) cycling [6], which is reflected as gradually diminishing memory cell storage noise margin (or increasing raw storage bit error rate). To meet a specified PE cycling endurance limit, NAND flash memory manufacturers must fabricate enough number of redundant memory cells that can tolerate the worst-case raw storage reliability at the end of memory lifetime. Clearly, the memory cell wear-out dynamics tend to make the existing worst-case oriented ECC redundancy largely *under-utilized* over the entire lifetime of memory, especially at its early lifetime when PE cycling number is relatively small.

Very intuitively, we may adaptively trade such under-utilized ECC redundancy for improving certain NAND flash memory system performance metrics throughout the memory lifetime. This naturally leads to a PE-cycling-aware adaptive NAND flash memory system design paradigm. Based upon extensive open literature on flash memory devices, we first develop an approximate NAND flash memory device model that quantitatively captures the dynamic PE cycling effects, including random telegraph noise [15, 17] and interface trap recovery and electron detrapping [26, 31, 45], and another major noise source: cell-to-cell interference [25]. Such a device model makes it possible to explores and quantitatively evaluate possible adaptive system design strategies. In particular, this paper explores the adaptive system design space from two perspectives: (1) Since NAND flash

[*] This material is based upon work supported by the National Science Foundation under Grant No. 0937794

memory program speed also strongly affects the memory cell storage noise margin, we could trade the under-utilized ECC redundancy to adaptively improve NAND flash memory program speed; (2) We could also exploit the under-utilized ECC redundancy to realize stronger memory cell defect tolerance and hence enable more aggressive technology scaling. We elaborate on the underlying rationale and realizations of these two design approaches. In addition, for the latter one, we propose a simple differential wear-leveling strategy in order to minimize its impact on effective PE cycling endurance.

For the purpose of evaluation, using the developed NAND flash memory device model, we first obtain detailed quantitative memory cell characteristics under different PE cycling times and different program speed for a hypothetical 2bit/cell NAND flash memory. Accordingly, with the sector size of 512B user data, we construct a binary BCH code (4798, 4096, 54) with 1.1% coding redundancy that can ensure the data storage integrity at the PE cycling limit of 10K. Using representative workload traces and the SSD model [3] in DiskSim [8], we carry out extensive simulations to evaluate the potential of trading under-utilized ECC redundancy to improve memory program speed while assuming the memory is defect-free. The simulation results show that we could reduce the SSD average response time by up to 32%. Assuming memory defects follow Poisson distributions, we further show that the proposed differential wear-leveling technique can very effectively improve the effectiveness of allocating ECC redundancy for improving memory defect tolerance. Finally, we study the combined effects when we trade the under-utilized ECC redundancy to improve memory program speed and realize defect tolerance at the same time. DiskSim-based simulations show that, even in the presence of high defect rates, we can still achieve noticeable SSD average response time reduction.

2 Background

2.1 Memory Erase and Program Basics

Each NAND flash memory cell is a floating gate transistor whose threshold voltage can be configured (or programmed) by injecting certain amount of charges into the floating gate. Hence, data storage in NAND flash memory is realized by programming the threshold voltage of each memory cell into two or more non-overlapping voltage windows. Before one memory cell can be programmed, it must be erased (i.e., remove the charges in the floating gate, which sets its threshold voltage to the lowest voltage window). NAND flash memory uses Fowler-Nordheim (FN) tunneling to realize both erase and program [7], because FN tunneling requires very low current and hence enables high erase/program parallelism. It is well known that the threshold voltage of erased memory cells tends to have a wide Gaussian-like distribution [41]. Hence, we can approximately model the threshold voltage distribution of erased state as

$$p_e(x) = \frac{1}{\sigma_e\sqrt{2\pi}} e^{-\frac{(x-\mu_e)^2}{2\sigma_e^2}}, \tag{1}$$

where μ_e and σ_e are the mean and standard deviation of the erased state threshold voltage. Regarding memory program, a tight threshold voltage control is typically realized by using incremental step pulse program (ISPP) [6, 39], i.e., all the memory cells on the same word-line are recursively programmed using a program-and-verify approach with a stair case program word-line voltage V_{pp}. Let ΔV_{pp} denote the incremental program step voltage. For the k-th programmed state with the verify voltage $V_p^{(k)}$, ideally ISPP program results in a uniform threshold voltage distribution:

$$p_p^{(k)}(x) = \begin{cases} \frac{1}{\Delta V_{pp}}, & \text{if } V_p^{(k)} \le x \le V_p^{(k)} + \Delta V_{pp} \\ 0, & \text{else} \end{cases}. \tag{2}$$

Unfortunately, the above *ideal* memory cell threshold voltage distribution can be (significantly) distorted in practice, mainly due to PE cycling and cell-to-cell interference, which will be discussed in the remainder of this section.

2.2 Effects of PE Cycling

Flash memory PE cycling causes damage to the tunnel oxide of floating gate transistors in the form of charge trapping in the oxide and interface states [9, 30, 34], which directly results in threshold voltage shift and fluctuation and hence gradually degrades memory device noise margin. Major distortion sources include

1. Electrons capture and emission events at charge trap sites near the interface developed over PE cycling directly result in memory cell threshold voltage fluctuation, which is referred to as random telegraph noise (RTN) [15, 17];

2. Interface trap recovery and electron detrapping [26, 31,45] gradually reduce memory cell threshold voltage, leading to the data retention limitation.

Moreover, electrons trapped in the oxide over PE cycling make it difficult to erase the memory cells, leading to a longer erase time, or equivalently, under the same erase time, those trapped electrons make the threshold voltage of the erased state increase [4, 21, 27, 42]. Most commercial flash chips employ erase-and-verify operation to prevent the increase of erase state threshold voltage at the penalty of gradually longer erase time with PE cycling.

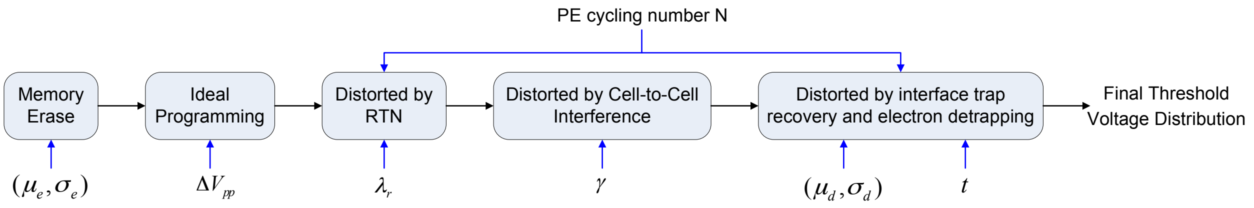


Figure 1: Illustration of the approximate NAND flash memory device model to incorporate major threshold voltage distortion sources.

RTN causes random fluctuation of memory cell threshold voltage, where the fluctuation magnitude is subject to exponential decay. Hence, we can model the probability density function $p_r(x)$ of RTN-induced threshold voltage fluctuation as a symmetric exponential function [15]:

$$p_r(x) = \frac{1}{2\lambda_r} e^{-\frac{|x|}{\lambda_r}}. \tag{3}$$

Let N denote the PE cycling number, λ_r scales with N in an approximate power-law fashion, i.e., λ_r is approximately proportional to N^α, where α tends to be less than 1.

Interface trap recovery and electron detrapping processes approximately follow Poisson statistics [30], hence threshold voltage reduction due to interface trap recovery and electron detrapping can be approximately modeled as a Gaussian distribution $\mathcal{N}(\mu_d, \sigma_d^2)$. Both μ_d and σ_d^2 scale with N in an approximate power-law fashion, and scale with the retention time t in a logarithmic fashion. Moreover, the significance of threshold voltage reduction induced by interface trap recovery and electron detrapping is also proportional to the initial threshold voltage magnitude [27], i.e., the higher the initial threshold voltage is, the faster the interface trap recovery and electron detrapping occur and hence the larger threshold voltage reduction will be.

2.3 Cell-to-Cell Interference

In NAND flash memory, the threshold voltage shift of one floating gate transistor can influence the threshold voltage of its neighboring floating gate transistors through parasitic capacitance-coupling effect [25]. This is referred to as cell-to-cell interference, which has been well recognized as the one of major noise sources in NAND flash memory [24,29,36]. Threshold voltage shift of a victim cell caused by cell-to-cell interference can be estimated as [25]

$$F = \sum_k (\Delta V_t^{(k)} \cdot \gamma^{(k)}), \tag{4}$$

where $\Delta V_t^{(k)}$ represents the threshold voltage shift of one interfering cell which is programmed after the victim cell, and the coupling ratio $\gamma^{(k)}$ is defined as

$$\gamma^{(k)} = \frac{C^{(k)}}{C_{total}}, \tag{5}$$

where $C^{(k)}$ is the parasitic capacitance between the interfering cell and the victim cell, and C_{total} is the total capacitance of the victim cell. Cell-to-cell interference significance is affected by NAND flash memory bit-line structure. In current design practice, there are two different bit-line structures, including conventional even/odd bit-line structure [35,40] and emerging all-bit-line structure [10,28]. For write, all-bit-line structure writes all the cells on the same wordline. In even/odd bit-line structure, memory cells on one word-line are alternatively connected to even and odd bit-lines and they are programmed at different time. Therefore, an even cell is mainly interfered by five neighboring cells and an odd cell is interfered by only three neighboring cells. Therefore even cells and odd cells experience largely different amount of cell-to-cell interference. Cells in all-bit-line structure suffers less cell-to-cell inference than even cells in odd/even structure, and the all-bit-line structure can most effectively support high-speed current sensing to improve the memory read and verify speed. Therefore, throughout the remainder of this paper, we mainly consider NAND flash memory with the all-bit-line structure.

2.4 An Approximate NAND Flash Memory Device Model

Based on the above discussions, we can approximately model NAND flash memory device characteristics as shown in Fig. 1. Accordingly, we can simulate memory cell threshold voltage distribution and the corresponding memory cell raw storage reliability. Based upon Eq.(1) and Eq.(2), we can obtain the distortion-less threshold voltage distribution function $p_p(x)$. Recall that $p_{pr}(x)$ denotes the RTN distribution function (see Eq.(3)), and let $p_{ar}(x)$ denote the threshold voltage distribution after incorporating RTN, which is obtained by convoluting $p_p(x)$ and $p_r(x)$:

$$p_{ar}(x) = p_p(x) \bigotimes p_r(x). \tag{6}$$

Cell-to-cell interference is further incorporated based on Eq.(4). To capture the inevitable process variability, we set both the vertical coupling ratio γ_y and diagonal coupling ratio γ_{xy} are random variables with bounded Gaussian distributions:

$$p_c(x) = \begin{cases} \frac{c_c}{\sigma_c\sqrt{2\pi}} \cdot e^{-\frac{(x-\mu_c)^2}{2\sigma_c^2}}, & \text{if } |x-\mu_c| \le w_c \\ 0, & \text{else} \end{cases}, \tag{7}$$

where μ_c and σ_c are the mean and standard deviation, and c_c is chosen to ensure the integration of this bounded Gaussian distribution equals to 1. We set $w_c = 0.1\mu_c$ and $\sigma_c = 0.4\mu_c$ in this work.

Let p_{ac} denote the threshold voltage distribution after incorporating cell-to-cell interference, $p_t(x)$ denote the distribution of threshold voltage fluctuation induced by interface trap recovery and electron detrapping, the final threshold voltage distribution p_f is obtained as

$$p_f(x) = p_{ac}(x) \bigotimes p_t(x). \tag{8}$$

Example 2.1 *Let us consider 2bits/cell NAND flash memory. We set normalized σ_e and μ_e of the erased state as 0.35 and 1.4, respectively. For the three programmed states, we set the normalized program step voltage ΔV_{pp} as 0.3, and the normalized verify voltages V_p as 2.85, 3.55 and 4.25, respectively. For the RTN distribution function $p_r(x)$, we set the parameter $\lambda_r = K_\lambda \cdot N^{0.5}$ where K_λ equals to 4×10^{-4}. Regarding cell-to-cell interference, according to [36, 38], we set the means of γ_y and γ_{xy} as 0.08 and 0.0048, respectively. For the function $\mathcal{N}(\mu_d, \sigma_d^2)$ to capture interface trap recovery and electron detrapping, according to [30, 31], we set that μ_d scale with $N^{0.5}$ and σ_d^2 scales with $N^{0.6}$, and both scale with $\ln(1+t/t_0)$, where t denotes the memory retention time and t_0 is an initial time and can be set as 1 hour. In addition, as pointed out earlier, both μ_d and σ_d^2 also depend on the initial threshold voltage. Hence, we set that both approximately scale with $K_s(x-x_0)$, where x is the initial threshold voltage, and x_0 and K_s are constants. Therefore, we have*

$$\begin{cases} \mu_d = K_s(x-x_0)K_d N^{0.5} \ln(1+t/t_0) \\ \sigma_d^2 = K_s(x-x_0)K_m N^{0.6} \ln(1+t/t_0) \end{cases}, \tag{9}$$

where we set $K_s = 0.333$, $x_0 = 1.4$, $K_d = 4 \times 10^{-4}$, and $K_m = 2 \times 10^{-6}$ by fitting the measurement data presented in [30, 31]. Accordingly, we carry out Monte Carlo computer simulations to obtain the cell threshold voltage distribution as shown in Fig. 2, which illustrates how RTN, cell-to-cell interference, and retention noise affect the threshold voltage distribution.

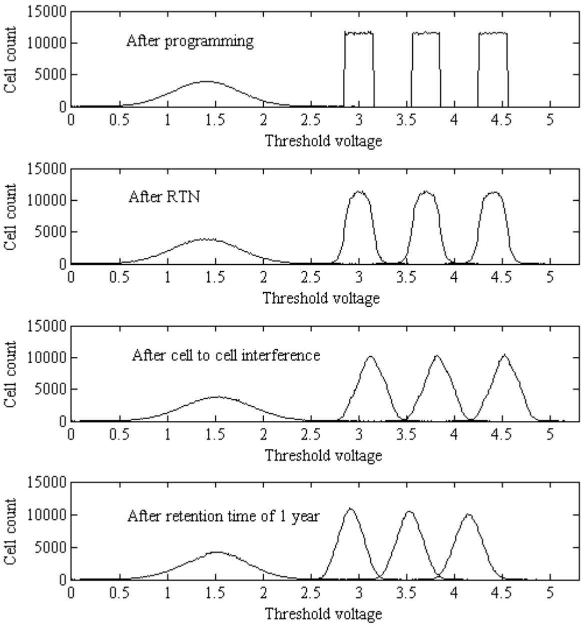


Figure 2: Simulated results to show the effects of RTN, cell-to-cell interference, and retention noise on memory cell threshold voltage distribution.

3 System Design Adaptive to PE Cycling

From the above discussions, it is clear that NAND flash memory cell raw storage reliability gradually degrades with the PE cycling: During the early lifetime of memory cells (i.e., the PE cycling number N is relatively small), the aggregated PE cycling effects are relatively small, which leads to a relatively large memory cell storage noise margin and hence good raw storage reliability (i.e., low raw storage bit error rate); since the aggregated PE cycling effects scale with N in approximate power-law fashions, the memory cell storage noise margin and hence raw storage reliability gradually degrade as the PE cycling number N increases. Given the target PE cycling endurance limit (e.g., 10K PE cycling), each memory word-line must have enough redundant memory cells so that the corresponding ECC can ensure the storage integrity as the PE cycling reaches the endurance limit. Due to the memory cell raw storage reliability dynamics, the redundancy geared to the worst-case scenario will *over-protect* the user data for most time throughout the entire memory lifetime, especially at its early lifetime when memory cell operational noise margin is much larger. This can be illustrated in Fig. 3, which clearly suggests that the redundant memory cells are essentially *under-utilized* at the memory early lifetime.

Very intuitively, we may trade such under-utilized redundancy to improve certain memory system performance metrics, which should be carried out adaptive to

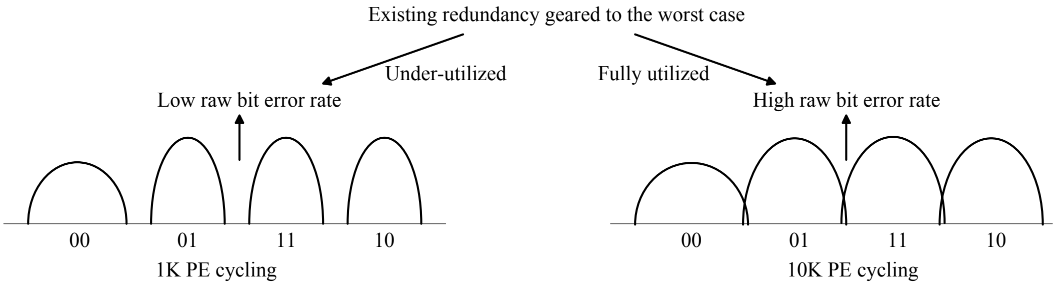


Figure 3: Illustration of the under-utilized ECC redundancy before reaching PE cycling endurance limit.

the memory PE cycling. In this work, we explore this adaptive memory system design space from two perspectives as discussed in the remainder of this section.

3.1 Approach I: Improve Memory Program Speed

In this subsection, we elaborate on the potential of trading the under-utilized ECC redundancy to improve average memory program speed. As discussed in Section 2.1, NAND flash memory program is carried out recursively by sweeping over the entire memory cell threshold voltage range with a program step voltage ΔV_{pp}. As a result, the memory program latency is inversely proportional to ΔV_{pp}, which suggests that we can improve the memory program speed by increasing ΔV_{pp}. However, a larger ΔV_{pp} directly results in a wider threshold voltage distribution of each programmed state, leading to less noise margin between adjacent programmed states and hence worse raw storage bit error rate. Therefore, there is an inherent trade-off between memory program speed vs. memory raw bit error rate, which can be configured by adjusting the program step voltage ΔV_{pp}. Since the memory cell noise margin is further degraded by the PE cycling effects as discussed above, a given ΔV_{pp} will result in different noise margin (hence different raw storage bit error rate) as memory cells undergo different amount of PE cycling.

In current design practice, ΔV_{pp} is fixed and its value is sufficiently small so that the ECC can tolerate the worst-case memory raw storage bit error rate as the PE cycling reaches its endurance limit. As a result, the memory program speed remains largely unchanged while the raw storage bit error rate gradually degrades. Before the PE cycling number reaches its endurance limit, the existing redundancy is under-utilized as pointed out in the above. Clearly, to eliminate such redundancy under-utilization, we could intentionally increase the the program step voltage ΔV_{pp} according to the run-time PE cycling number in such a way that the memory raw storage bit error rate is always close to what can be maximally tolerated by the existing redundancy. Therefore, the existing redundancy is always almost fully utilized, and meanwhile the dynamically increased ΔV_{pp} leads to higher average memory program speed. The above discussion can be further illustrated in Fig. 4.

Although it would be ideal if the program step voltage ΔV_{pp} can be smoothly adjusted with a very fine granularity, the limited reference voltage accuracy in real NAND flash memory chips may only enable the use of a few discrete program step voltages. Assume there are m different program step voltages, i.e., $\Delta V_{pp}^{(1)} > \Delta V_{pp}^{(2)} > \cdots > \Delta V_{pp}^{(m)}$. Given the existing ECC redundancy, we can obtain a sequence of PE cycling thresholds $N_0 = 0 < N_1 < \cdots < N_m$ so that, if the run-time PE cycling number falls into the range of $[N_{i-1}, N_i)$, we can use the program step voltage $V_{pp}^{(i)}$ and still ensure the overall system data storage integrity. If we follow the conventional design practice where the program step voltage is fixed according to the worst-case scenario, the smallest step voltage $\Delta V_{pp}^{(m)}$ will be used throughout the entire memory lifetime. Therefore, we can estimate the average program speed improvement over the entire memory lifetime as

$$s = 1 - \frac{\sum_{i=1}^{m}(N_i - N_{i-1}) \cdot \frac{1}{\Delta V_{pp}^{(i)}}}{N_m \cdot \frac{1}{\Delta V_{pp}^{(m)}}}. \tag{10}$$

3.2 Approach II: Improve Memory Technology Scalability

In this subsection, we elaborate on the potential of trading the under-utilized ECC redundancy to improve memory defect tolerance. With the help of very sophisticated techniques such as double patterning [20], the decade-long 193nm photolithography has successfully pushed NAND flash memory into the sub-30nm region. However, as the industry is striving to push the NAND flash memory technology scaling into the sub-20nm region by using immersion photolithography or new lithography technologies such as nanoimprint, defects in such extremely dense memory arrays may inevitably increase.

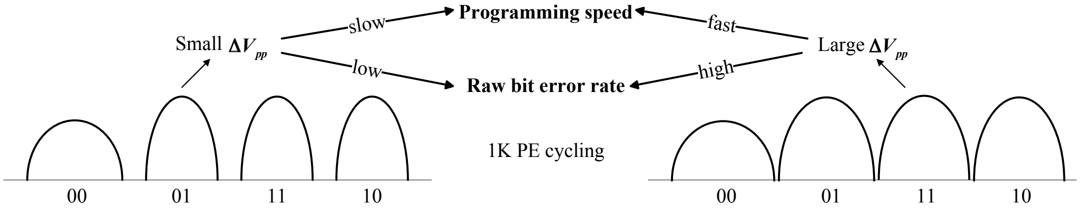


Figure 4: Illustration of the impact of program step voltage ΔV_{pp} on the program speed vs. raw storage bit error rate trade-off.

As a result, conventional spare row/column repair techniques may become inadequate to ensure a sufficiently high yield.

Very intuitively, the existing ECC redundancy can be leveraged to tolerate memory defects, especially random memory cell defects. However, if certain portion of ECC redundancy is used for defect tolerance, it will not be able to ensure the specified PE cycling limit, leading to PE cycling endurance degradation. Since all the pages in each memory block undergo the same number of PE cycling, the worst-case page (i.e., the page contains the most defects) in each block sets the achievable PE cycling endurance for this block. For example, assume the existing ECC redundancy can tolerate up to 50 errors for each page and survive up to 10K PE cycling in the absence of any memory cell defects. If the worst-case page in one block contains 5 defective cells, then it can only use the residual 45-error-correcting capability to tolerate memory operational noises such as PE cycling effects and cell-to-cell interference. Suppose this makes the worst-case page can only survive up to 8K PE cycling, this block can only be erased by 8K times instead of 10K times before risking data loss.

Clearly, if we attempt to reserve certain ECC redundancy for tolerating memory cell defects, we must minimize the impact on overall memory PE cycling endurance. In current design practice, NAND flash memory uses wear-leveling to uniformly spread program/erase operations among all the memory blocks to maximize the overall memory lifetime. Since different memory blocks with different amount of defective memory cells can survive different number of PE cycling, uniform wear-leveling is clearly not an optimal option. Instead, we should make wear-leveling fully aware of the different achievable PE cycling limits among different memory blocks, which is referred to as *differential wear-leveling*. This can be illustrated in Fig. 5: instead of uniformly distributing program/erase operations among all the memory blocks, the differential wear-leveling schedule the program/erase operations among all the memory blocks in proportional to their achievable PE cycling limits. As a result, we may largely improve the overall memory lifetime compared with uniform wear-leveling.

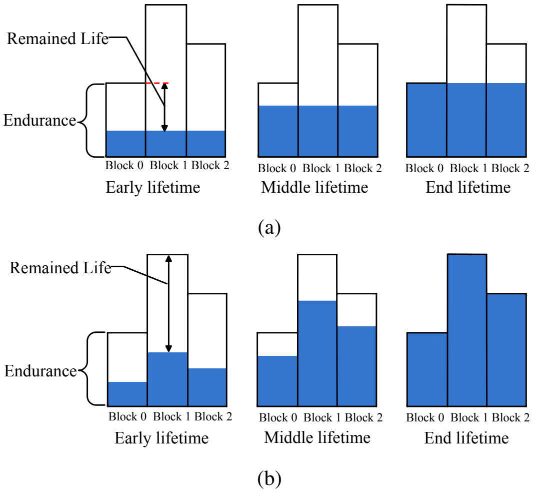


Figure 5: Illustration of (a) conventional uniform wear-leveling, and (b) proposed differential wear-leveling, where the ECC is used to tolerate defective memory cells and hence different blocks may have different achievable PE cycling endurance.

Assume the worst-case page can at most contains M defective memory cells, and let P_d denote the probability that the worst-case page in one block contains $d \in [0, M]$ defective memory cells. Given the number of defective memory cells in the worst-case page d, we can obtain the corresponding achievable PE cycling endurance limit $N^{(d)}$, i.e., the ECC can ensure a PE cycling number up to $N^{(d)}$ while tolerating d defective memory cells. Clearly, we have $N^{(0)} > N^{(1)} > \cdots > N^{(M)}$, where $N^{(0)}$ is the achievable PE cycling limit in the defect-free scenario. Define the *effective PE cycling endurance* as the average PE cycling limits of all the memory blocks. Under the uniform wear-leveling, the memory chip can only sustain PE cycling of $N^{(M)}$. Therefore, compared with the defect-free scenario, the effective PE cycling endurance degrades by $N^{(0)}/N^{(M)}$, which can result in a significant memory lifetime degradation. On the other hand, under the ideal differential wear-leveling, each block can reach its own PE cycling limit as illustrated in Fig. 5, hence the effective PE cycling endurance will be $\sum_{d=0}^{M} P_d \cdot N^{(d)}$, representing the improvement of

$$\frac{\sum_{d=0}^{M} P_d \cdot N^{(d)}}{N^{(M)}} \tag{11}$$

over the uniform wear-leveling. We note that this design approach can be combined with the one presented in Section 3.1 to improve average memory program speed in the presence of memory cell defects. Given the number of defective memory cells d and the set of m program step voltage $\Delta V_{pp}^{(i)}$ for $1 \leq i \leq m$, we can obtain a set of PE cycling thresholds $N_0^d = 0 < N_1^d < \cdots < N_m^d$, i.e., if present PE cycling number falls into the range of $[N_{i-1}^{(d)}, N_i^{(d)})$, we can use the program step voltage $\Delta V_{pp}^{(i)}$ and meanwhile ensure the tolerance to d defective memory cells. Therefore, for the blocks whose worst-case page contains d defective memory cells, the average program speed improvement is

$$s_d = 1 - \frac{\sum_{i=1}^{m} (N_i^d - N_{i-1}^d) \cdot \frac{1}{\Delta V_{pp}^{(i)}}}{N_m^d \cdot \frac{1}{\Delta V_{pp}^{(m)}}}. \tag{12}$$

The overall average program speed improvement can be further estimated as $\sum_{i=1}^{M} P_i \cdot s_i$.

4 Evaluation Results

We carried out simulations and analysis to future demonstrate the effectiveness of the above two simple design approaches and their combination. To carry out trace-based simulations, we use the SSD module [3] in DiskSim [8], and use 6 workload traces including Iozone and Postmark [3], Finance1 and Finance2 from [1], and Trace1 and Trace2 from [16]. The simulator can support the use of several parallel packages that can work in parallel to improve the SSD throughput. Each package contains 2 dies that share an 8-bit I/O bus and a number of common control signals, and each die contains 4 planes and each plane contains 2048 blocks. Each block contains 64 4KB pages, each of which consists of 8 512B sectors. Following the version 2.1 of the Open NAND Flash Interface (ONFI) [2], we set the NAND flash chip interface bus frequency as 200MB/s. Regarding the ECC, we assume that binary (n, k, t) BCH codes are being used, where n is the codeword length, k is the user data length (i.e., 512B in this study), and t is the error-correcting capability. We consider the use of 2bit/cell NAND flash memory, and set the baseline 2bit/cell NAND flash memory using the equivalent memory channel model parameters presented in Example 2.1 in Section 2.4, for which a (4798, 4096, 54) BCH code can ensure a PE cycling endurance limit of 10K under the retention time of 1 year. We note that the target NAND flash memory retention time is fixed as 1 year throughout all the studies in this work.

In this section, we first present trace-based simulation results to demonstrate how the first design approach can reduce the overall request response time and hence improve SSD speed performance. Then, we present analysis results to demonstrate the second design approach by assuming memory cell defects follow Poisson distribution. Finally, we demonstrate the effectiveness when these two approaches are combined together to improve SSD speed performance in the presence of memory cell defects.

4.1 Improve SSD Speed Performance

In the baseline scenario with the parameters listed in Example 2.1, the normalized program step voltage ΔV_{pp} is 0.3. As discussed in Section 3.1, we can use larger-than-worst-case ΔV_{pp} over the memory lifetime to improve memory program speed by exploiting the memory device wear-out dynamics. In this work, we assume that memory chip voltage generators can increase ΔV_{pp} with a step of 0.05, hence we consider four different normalized values: $\Delta V_{pp}^{(1)} = 0.45$, $\Delta V_{pp}^{(2)} = 0.4$, $\Delta V_{pp}^{(1)} = 0.35$, and $\Delta V_{pp}^{(4)} = 0.3$. By carrying out Monte Carlo simulations without changing the other memory model parameters, we have that these four different program step voltages can survive up to $N_1 = 2710$, $N_2 = 4820$, $N_3 = 7500$, and $N_4 = 10000$ PE cycling, respectively, under the retention time of 1 year. Therefore, according to Eq.(10), the average NAND flash memory program speed can be improved by 18% compared with the baseline scenario. We further carried out DiskSim-based simulations to investigate how such improved memory program speed can reduce the SSD average response time (incorporating both write and read request response time) for different traces under different system configurations. We set that the 2bit/cell NAND flash memory program latency as 600μs when the normalized program step voltage ΔV_{pp} is 0.3, on-chip memory sensing latency as 30μs, and erase time as 3ms.

In this study, we consider the use of 4 and 8 parallel packages. Fig. 6 compares the normalized SSD average response time when using 4 and 8 parallel packages, respectively, where we set ΔV_{pp} as 0.3. It shows that using more parallel packages can directly improve SSD speed performance, which can be intuitively justified. Fig. 7(a) and Fig. 7(b) show the normalized SSD average response time under the 4 different normalized program step voltage ΔV_{pp} for all the 6 traces when the SSD contains 4 and 8 parallel packages, respectively. We use the first-come first-serve (FCFS) scheduling scheme in the simulations. Compared with the baseline scenario with $\Delta V_{pp} = 0.3$, the average response time can be reduced by up to ∼50%

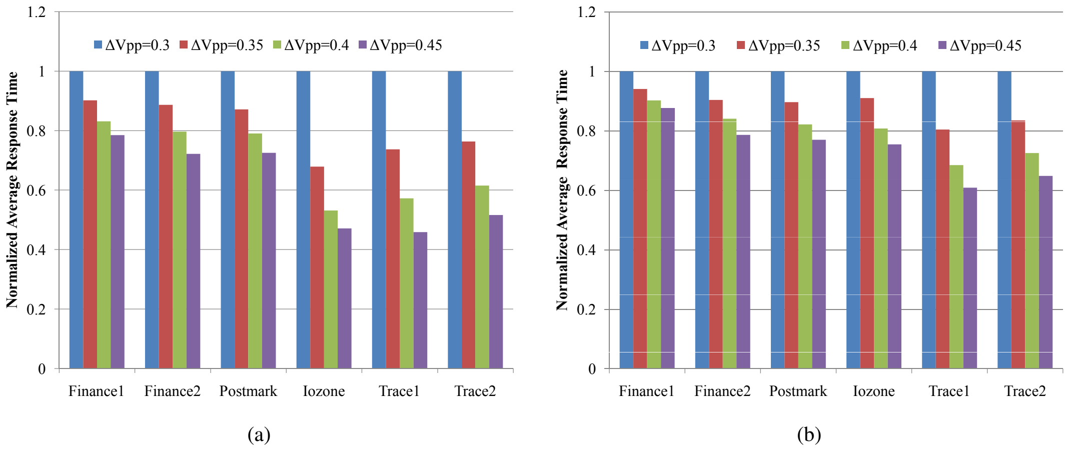


Figure 7: Simulated normalized average response time when the SSD contains (a) 4 parallel packages, and (b) 8 parallel packages.

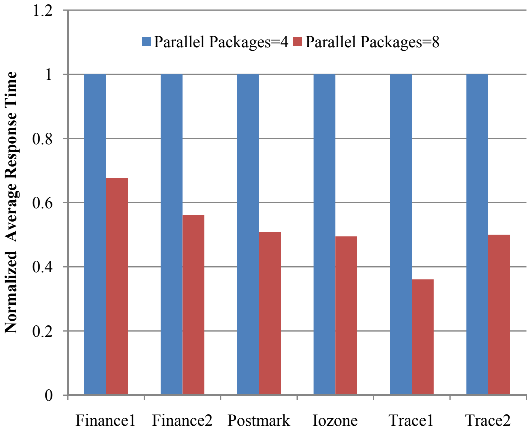


Figure 6: Comparison of normalized SSD average response time with 4 and 8 parallel packages ($\Delta V_{pp} = 0.3$).

with 4 parallel packages and up to ∼40% with 8 parallel packages. The results show that the use of larger program step voltage can consistently improve SSD speed performance under different number of parallel packages.

Given the PE cycling thresholds N_i for $i = 1,2,3,4$ as presented in the above, the NAND flash memory should employ the program step voltage $\Delta V_{pp}^{(i)}$ when the present PE cycling number falls into $[N_{i-1},\ N_i)$, where N_0 is set to 0. Therefore, based on the the simulation results shown in Fig. 7, we can obtain the overall SSD average response time reduction compared with the baseline scenario, as shown in Fig. 8. It shows that this proposed design approach can noticeably improve the overall SSD speed performance. Intuitively, those traces with higher write request ratios (e.g., Iozone, Trace1, and Trace2) tend to benefit more from this design approach, as shown in Fig. 8. In addition, as we increase the package parallelism from 4 to 8, the overall response time reduction consistently reduces over all the traces. This can be explained as follows: As the SSD contains more parallel packages, the increased architecture-level parallelism will directly improve SSD speed performance, as illustrated in Fig. 6. As a result, this will make the improvement on the device-level program speed become relatively less significant with respect to the improvement of overall system speed performance.

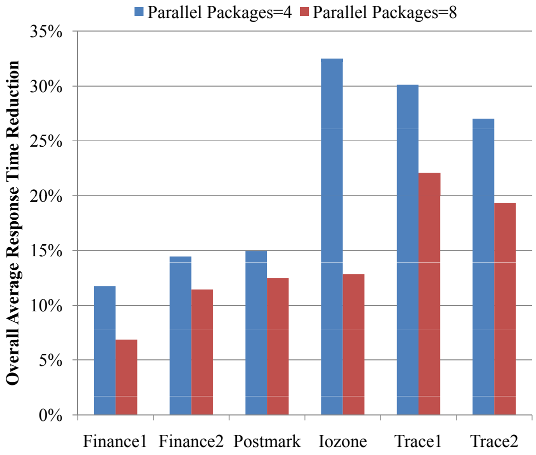


Figure 8: Overall SSD average response time reduction compared with the baseline scenario when using 4 and 8 parallel packages.

In the above simulations, the FCFS scheduling scheme has been used. To study the sensitivity of this design approach to different scheduling schemes, we repeat the above simulations using two other popular scheduling schemes including ELEVATOR and SSTF (shortest seek time first) [43]. Fig. 9 shows the overall SSD average response time reduction compared with the baseline scenario, where the SSD contains 4 parallel packages. The results show the proposed design approach can consistently improve overall SSD speed performance under different scheduling schemes.

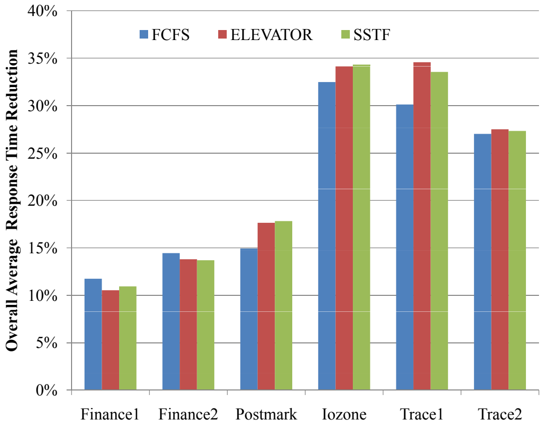


Figure 9: Overall SSD average response time reduction compared with the baseline scenario under different scheduling schemes.

4.2 Improve Defect Tolerance

To demonstrate the proposed design approach for improving memory defect tolerance, we assume that the number of defective memory cells in each worst-case page follows a Poisson distribution that is widely used to model defects in integrated circuits. Therefore, under the Poisson-based distribution model, the probability that the worst-case page in each block contains d defective memory cells is $f(k;\lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$, where the parameter λ is the mean of the number of defective memory cells in each worst-case page. Given the parameter λ, we find the value M so that $\sum_{i=0}^{M} f(i;\lambda) \geq 0.999$, and assume that any blocks whose worst-case page contains more than M defective memory cells can be replaced by a redundant block. In this work, we consider the mean λ ranging from 1 to 4, and accordingly have that the maximum value of M is 12.

Using the baseline NAND flash memory model parameters as listed in Example 2.1, we can obtain the achievable PE cycling limit $N^{(d)}$ for each d, i.e., we use the (4798, 4096, 54) BCH code to tolerate d defective memory cells and meanwhile use its residual $(54-d)$-error-correcting capability to ensure a PE cycling endurance limit of $N^{(d)}$ under the retention time of 1 year. Fig. 10 shows the achievable PE cycling limit $N^{(d)}$ with d ranging from 0 to 12. Under different value of mean λ, we have different value of M, denoted as $M^{(\lambda)}$. When the uniform wear-leveling is being used, the effective PE cycling endurance is simply $N^{(d)}$ when $d = M^{(\lambda)}$. When the proposed differential wear-leveling is being used, the effective PE cycling endurance is

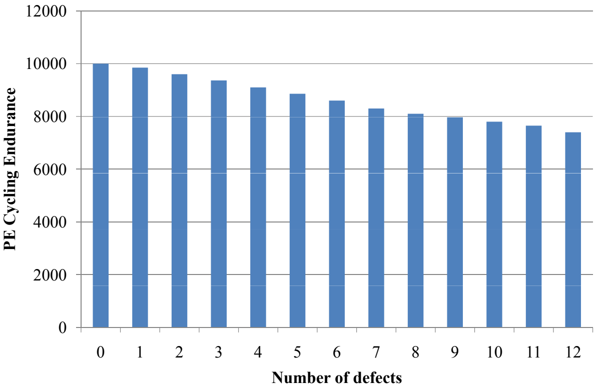


Figure 10: Achievable PE cycling endurance under different value of defective memory cells in the worst-case page.

$$\sum_{d=0}^{M^{(\lambda)}} \frac{\lambda^k e^{-\lambda}}{k!} \cdot N^{(d)}, \tag{13}$$

for a given mean λ. Fig. 11 shows the effective PE cycling endurance when these two different wear-leveling schemes are being used under different value of λ. The results show that the proposed differential wear-leveling can noticeably improve the effective PE cycling endurance and hence SSD lifetime compared with uniform wear-leveling. As the defects density increases (i.e., λ

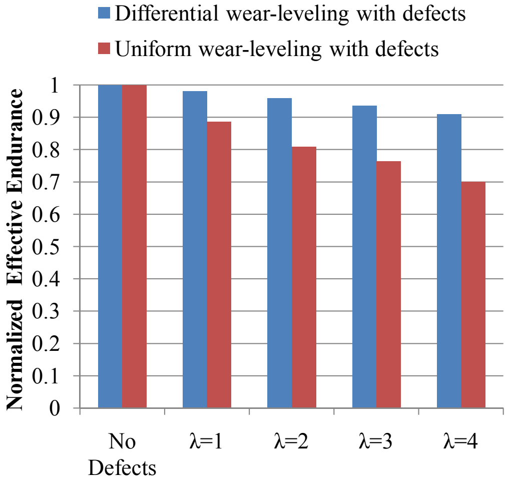


Figure 11: Effective PE cycling endurance when using uniform wear-leveling and differential wear-leveling under different value of λ.

increases), the gain of differential wear-leveling over uniform wear-leveling will accordingly improve (i.e., from about 10% improvement at $\lambda = 1$ to about 30% improvement at $\lambda = 4$).

4.3 Combination of the Two Design Approaches

As discussed earlier, we can combine the proposed two design approaches in order to improve SSD speed performance when ECC is also used to tolerate defective memory cells. Following the discussions in Section 4.2, we assume that the number of defective memory cells in the worst-case page has a Poisson distribution and consider the cases when the mean λ ranges from 1 to 4. Following the discussions in Section 4.1, beyond the normalized program step voltage ΔV_{pp} of 0.3 in the baseline scenario, we consider three larger values of ΔV_{pp}, including 0.35, 0.4, and 0.45. Denote $\Delta V_{pp}^{(1)} = 0.45$, $\Delta V_{pp}^{(2)} = 0.4$, $\Delta V_{pp}^{(3)} = 0.35$, and $\Delta V_{pp}^{(4)} = 0.3$. Given the memory cell defects number d and the (4798, 4096, 54) BCH code being used, we can obtain a set of PE cycling thresholds $N_0^d = 0 < N_1^d < \cdots < N_4^d$ so that, if present PE cycling number falls into the range of $[N_{i-1}^{(d)}, N_i^{(d)})$, we can use the program step voltage $\Delta V_{pp}^{(i)}$ and meanwhile ensure the tolerance of d defective memory cells. Fig. 12 shows the PE cycling thresholds when the defect number increases from 0 to 12. The results can be intuitively justified: as the defect number increases, the residual ECC error-correcting capability degrades, and consequently the larger program step voltage can only be used over a less number of PE cycling.

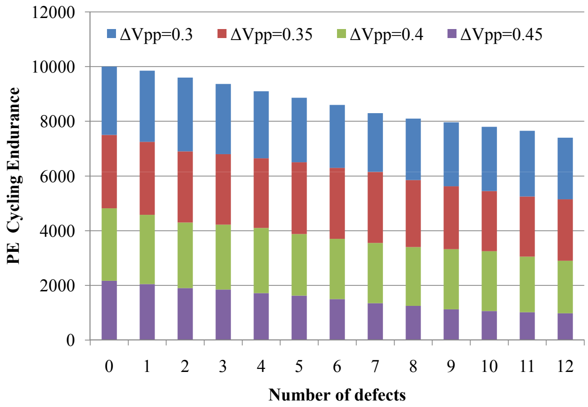


Figure 12: PE cycling thresholds corresponding to different number of defective cells in the worst-case page of one block.

Given each program step voltage $\Delta V_{pp}^{(i)}$, we can obtain the normalized SSD response time τ_i for each specific trace, as shown in Fig. 7. Recall that, when the PE cycling number falls into the range $[N_{i-1}^{(d)}, N_i^{(d)})$, we can use the program step voltage $\Delta V_{pp}^{(i)}$, and the baseline scenario fixes the program step voltage as $\Delta V_{pp}^{(4)} = 0.3$ throughout the entire memory lifetime. Therefore, we can calculate the overall SSD average response time reduction over the baseline scenario for each trace as

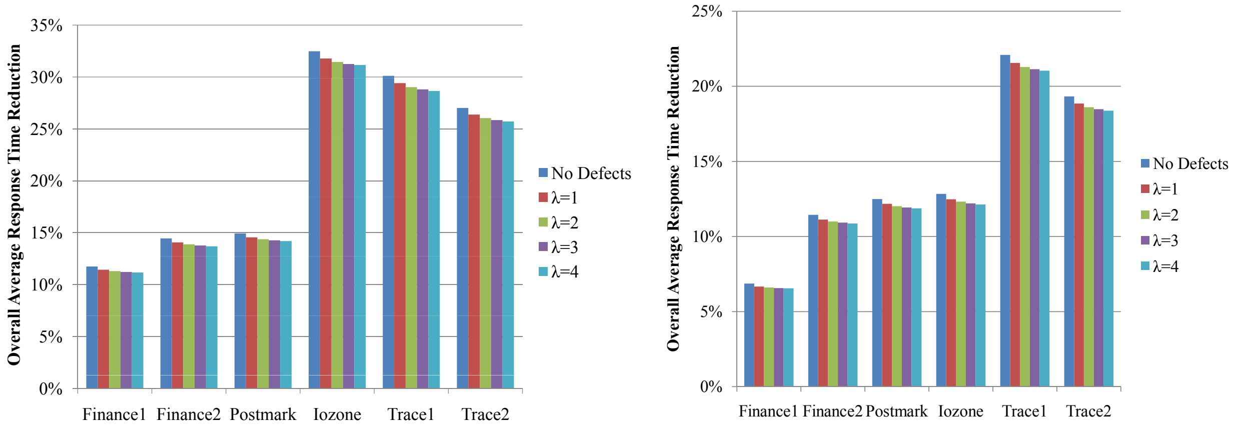
$$\sum_{d=0}^{12} f(d;\lambda) \frac{\sum_{i=1}^{4} (N_i^{(d)} - N_{i-1}^{(d)}) \cdot \tau_i}{N_4^{(d)} \cdot \tau_4}, \qquad (14)$$

and the results are shown in Fig. 13. The results suggest that we still can maintain a noticeable SSD speed performance improvement when ECC is also used to tolerate defective memory cells.

5 Related Work

NAND flash memory system design has attracted many recent attentions, where most work focused on improving system speed performance and endurance. Dirik and Jacob [16] studied the effect on SSD system speed performance by changing various SSD system parallelism and concurrency at different levels such as the numbers of planes on each channel and the number of channels, and compared various existing disk access scheduling algorithms. Agrawal *et al.* [3] analyzed the effect of page size, striping and interleaving policy on the memory system performance, and proposed a conception of *gang* as a higher-level "superblock" to facilitate SSD system-level parallelism configurations. Min and Nam [32] developed several NAND flash memory performance enhancement techniques such as write request interleaving. Seong *et al.* [37] applied bus-level and chip-level interleaving to exploit the inherent parallelism in multiple flash memory chips to improve the SSD speed performance. The authors of [11,13] applied adaptive bank scheduling policies to achieve an even distribution of write request and load balance to improve system speed performance.

Wear-leveling is used to improve NAND flash memory endurance. Gal and Toledo [18] surveyed many patented and published wear-leveling algorithms and data structures for NAND flash memory. Ben-Aroya and Toledo [5] more quantitatively evaluated different wear-leveling algorithms, including both on-line and off-line algorithms. The combination of wear-leveling and garbage collection and the involved design tradeoffs have been investigated by many researchers, e.g., see [12, 14, 22, 23, 44]. In current design practice, defect tolerance has been mainly realized by bad block management that run-time monitors and disables the future use of blocks with defects. Traditional redundant repair can also be used to compensate certain memory defects,



(a)

(b)

Figure 13: Overall average response time reduction over the baseline scenario under different λ when SSD contains (a) 4 parallel packages and (b) 8 parallel packages.

e.g., see [19]. In addition, a NAND flash memory device model was presented in [33], which nevertheless does not take into account of RTN noise and cell-to-cell interference, and the model was used to show that time-dependent trap recovery can be leveraged to improve memory endurance.

We note that most prior work on improving SSD system speed performance and/or memory endurance are carried out mainly from architecture/system perspective to combat flash memory device issues. To the best of our knowledge, this paper represents the first attempt to adaptively exploit flash memory device characteristics, in particular PE-cycling-dependent device wear-out dynamics, at the system level to improve SSD system speed performance and NAND flash memory scalability. The proposed design approaches are completely orthogonal to prior architecture/system level techniques and can be readily combined together.

6 Conclusion

This paper investigates the potential of adaptively leveraging NAND flash memory cell wear-out dynamics to improve memory system performance. As memory PE cycling increases, NAND flash memory cell storage noise margin and hence raw storage reliability accordingly degrade. Therefore, the specified PE cycling endurance limit determines the worst-case raw memory storage reliability, which further sets the amount of redundant memory cells that must be fabricated. Motivated by the fact that such worst-case oriented redundancy is essentially under-utilized over the entire memory lifetime, especially when the PE cycling number is relatively small, this paper proposes to trade such under-utilized redundancy to improve system speed performance and/or tolerate defective memory cells. We further propose a simple differential wear-leveling scheme to minimize the impact on PE cycling endurance if the redundancy is used to tolerate defective memory cells. To quantitatively evaluate such adaptive NAND flash memory system design strategies, we first develop an approximate NAND flash memory device model that can capture the effects of PE cycling on memory cell storage reliability. To evaluate the effectiveness on improving memory system speed, we carry out extensive simulations over a variety of traces using the DiskSim-based SSD simulator under different system configurations, and the results show up to 32% SSD average response time reduction can be achieved. To evaluate the effectiveness on defect tolerance, with a Poisson-based defect statics model, we show that this design strategy can tolerate relatively high defect rates at small degradation of effective PE cycling endurance. Finally, we show that these two aspects can be combined together so that we could noticeably reduce SSD average response time even in the presence of high memory defect densities. generate the the references with alphatical order.

Acknowledgments

We thank Albert Fazio at Intel for his valuable comments on NAND flash memory device modeling. We also thank the anonymous reviewers and our shepherd Eno Thereska for their feedback.

References

[1] “SPC Trace File Format Specification. http://traces.cs.umass.edu/index.php/Storage/Storage,

Last accessed on June 6, 2010," Storage Performance Council, Tech. Rep. Revision 1.0.1, 2002.

[2] "Open NAND Flash Interface Specification," Hynix Semiconductor and Intel Corporation and Micron Technology, Inc. and Numonyx and Phison Electronics Corp. and Sony Corporation and Spansion, Tech. Rep. Revision 2.1, Jan. 2009.

[3] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design Tradeoffs for SSD Performance," in *Proc. of USENIX Annual Technical Conference*, 2008, pp. 57–70.

[4] S. Aritome, R. Shirota, G. Hemink, T. Endoh, and F. Masuoka, "Reliability Issues of Flash Memory Cells," *Proceedings of the IEEE*, vol. 81, no. 5, pp. 776–788, 1993.

[5] A. Ben-Aroya and S. Toledo, "Competitive Analysis of Flash-Memory Algorithms," in *Proc. of the Annual European Symposium*, 2006, pp. 100–111.

[6] R. Bez, E. Camerlenghi, A. Modelli, and A. Visconti, "Introduction to Flash memory," *Proceedings of the IEEE*, vol. 91, pp. 489–502, April 2003.

[7] R. Bez and P. Cappelletti, "Flash Memory and Beyond," in *Proc. of IEEE VLSI-TSA International Symposium on VLSI Technology*, 2005, pp. 84–87.

[8] J. S. Bucy, J. Schindler, S. W. Schlosser, and G. R. Ganger, "The DiskSim Simulation Environment Version 4.0 Reference Manual," Carnegie Mellon University Parallel Data Lab, Tech. Rep. CMU-PDL-08-101, May 2008.

[9] P. Cappelletti, R. Bez, D. Cantarelli, and L. Fratin, "Failure Mechanisms of Flash Cell in Program/erase Cycling," in *Proc. of International Electron Devices Meeting (IEDM)*, 1994, pp. 291–294.

[10] R.-A. Cernea and et al., "A 34 MB/s MLC Write Throughput 16 Gb NAND With All Bit Line Architecture on 56 nm Technology," *IEEE J. Solid-State Circuits*, vol. 44, pp. 186–194, Jan. 2009.

[11] L.-P. Chang and T.-W. Kuo, "An Adaptive Striping Architecture for Flash Memory Storage Systems of Embedded Systems," *Proc. of IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 187 – 196, 2002.

[12] L.-P. Chang, T.-W. Kuo, and S.-W. Lo, "Real-time Garbage Collection for Flash-Memory Storage Systems of Real-time Embedded Systems," *ACM Transactions on Embedded Computing Systems*, vol. 3, no. 4, pp. 837–863, 2004.

[13] Y.-B. Chang and L.-P. Chang, "A Self-Balancing Striping Scheme for NAND-Flash Storage Systems," in *Proc. of the ACM Symposium on Applied Computing*, 2008, pp. 1715–1719.

[14] M.-L. Chiang and R.-C. Chang, "Cleaning Policies in Mobile Computers Using Flash Memory," *Journal of Systems and Software*, vol. 48, no. 3, pp. 213–231, 1999.

[15] C. Compagnoni, M. Ghidotti, A. Lacaita, A. Spinelli, and A. Visconti, "Random Telegraph Noise Effect on the Programmed Threshold-Voltage Distribution of Flash Memories," *IEEE Electron Device Letters*, vol. 30, no. 9, 2009.

[16] C. Dirik and B. Jacob, "The Performance of PC Solid-state disks (SSDs) as a Function of Bandwidth, Concurrency, Device Architecture, and System Organization," *SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 279–289, 2009.

[17] K. Fukuda, Y. Shimizu, K. Amemiya, M. Kamoshida, and C. Hu, "Random Telegraph Noise in Flash Memories - Model and Technology Scaling," in *Proc. of IEEE International Electron Devices Meeting (IEDM)*, 2007, pp. 169–172.

[18] E. Gal and S. Toledo, "Algorithms and Data Structures for Flash Memories," *ACM Computing Surveys*, vol. 37, no. 2, pp. 138–163, 2005.

[19] Y.-Y. Hsiao, C.-H. Chen, and C.-W. Wu, "Built-In Self-Repair Schemes for Flash Memories," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 8, pp. 1243 –1256, Aug. 2010.

[20] B. Hwang and et al, "Comparison of Double Patterning Technologies in NAND Flash Memory With Sub-30nm Node," in *Proc. of the European Solid State Device Research Conference (ESSDERC)*, 2009, pp. 269 –271.

[21] T. Jung and et al., "A 117-mm 3.3-V only 128-Mb Multilevel NAND Flash Memory for Mass Storage Applications," *IEEE J. Solid-State Circuits*, vol. 31, no. 11, pp. 1575–1583, Nov. 1996.

[22] A. Kawaguchi, S. Nishioka, and H. Motoda, "A Flash-Memory Based File System," in *Proc. of the USENIX Technical Conference*, 1995, pp. 13–13.

[23] H. Kim and S. Lee, "An Effective Flash Memory Manager for Reliable Flash Memory Space Management," *IEICE Transactions on Information and Systems*, vol. 85, no. 6, pp. 950–964, 2002.

[24] K. Kim and et.al, "Future Memory Technology: Challenges and Opportunities," in *Proc. of International Symposium on VLSI Technology, Systems and Applications*, Apr. 2008, pp. 5–9.

[25] J.-D. Lee, S.-H. Hur, and J.-D. Choi, "Effects of Floating-Gate Interference on NAND Flash Memory Cell Operation," *IEEE Electron Device Letters*, vol. 23, no. 5, pp. 264–266, May. 2002.

[26] J. Lee, J. Choi, D. Park, and K. Kim, "Data Retention Characteristics of sub-100 nm NAND Flash Memory Cells," *IEEE Electron Device Letters*, vol. 24, no. 12, pp. 748–750, 2003.

[27] J. Lee, J. Choi, D. Park, K. Kim, R. Center, S. Co, and S. Gyunggi-Do, "Effects of Interface Trap Generation and Annihilation on the Data Retention Characteristics of Flash Memory Cells," *IEEE Transactions on Device and Materials Reliability*, vol. 4, no. 1, pp. 110–117, 2004.

[28] Y. Li and et. al, "A 16Gb 3b/Cell NAND Flash Memory in 56nm with 8MB/s Write Rate," in *Proc. of IEEE International Solid-State Circuits Conference (ISSCC)*, Feb. 2008, pp. 506–632.

[29] H. Liu, S. Groothuis, C. Mouli, J. Li, K. Parat, and T. Krishnamohan, " 3D Simulation Study of Cell-Cell Interference in Advanced NAND Flash Memory," in *Proc. of IEEE Workshop on Microelectronics and Electron Devices*, April 2009.

[30] N. Mielke, H. Belgal, I. Kalastirsky, P. Kalavade, A. Kurtz, Q. Meng, N. Righos, and J. Wu, "Flash EEPROM Threshold Instabilities Due to Charge Trapping During Program/erase Cycling," *IEEE Transactions on Device and Materials Reliability*, vol. 4, no. 3, pp. 335–344, 2004.

[31] N. Mielke, H. Belgal, A. Fazio, Q. Meng, and N. Righos, "Recovery Effects in the Distributed Cycling of Flash Memories," in *Proc. of IEEE International Reliability Physics Symposium*, 2006, pp. 29–35.

[32] S. L. Min and E. H. Nam, "Current Trends in Flash Memory Technology," *Proc. of Asia and South Pacific Conference on Design Automation.*, p. 2., Jan. 2006.

[33] V. Mohan, T. Siddiqua, S. Gurumurthi, and M. R. Stan, "How I Learned to Stop Worrying and Love Flash Endurance," in *Proc. of the 2nd USENIX conference on Hot topics in storage and file systems*, 2010, pp. 3–3.

[34] P. Olivo, B. Ricco, and E. Sangiorgi, "High Field Induced Voltage Dependent Oxide Charge," *Applied Physics Letter*, vol. 48, pp. 1135–1137, 1986.

[35] K.-T. Park and et al., "A Zeroing Cell-to-Cell Interference Page Architecture With Temporary LSB Storing and Parallel MSB Program Scheme for MLC NAND Flash Memories," *IEEE J. Solid-State Circuits*, vol. 40, pp. 919–928, Apr. 2008.

[36] K. Prall, "Scaling Non-Volatile Memory Below 30nm," in *Proc. of IEEE Non-Volatile Semiconductor Memory Workshop*, Aug. 2007, pp. 5–10.

[37] Y. J. Seong, E. H. Nam, J. H. Yoon, H. Kim, J. Choi, S. Lee, Y. H. Bae, J. Lee, Y. Cho, and S. L. Min, "Hydra: A Block-Mapped Parallel Flash Memory Solid-State Disk Architecture," *IEEE Transactions on Computers*, vol. 59, no. 7, pp. 905 –921, Jul. 2010.

[38] N. Shibata and et al., "A 70 nm 16 Gb 16-level-cell NAND flash memory," in *Proc. of IEEE Symposium on VLSI Circuits*, 2007, pp. 190–191.

[39] K.-D. Suh and et al., "A 3.3 V 32 Mb NAND Flash Memory with Incremental Step Pulse Programming Scheme," *IEEE J. Solid-State Circuits*, vol. 30, no. 11, pp. 1149–1156, Nov. 1995.

[40] K. Takeuchi and et al., "A 56-nm CMOS 99-mm^2 8-Gb Multi-Level NAND Flash Memory With 10-MB/s Program Throughput," *IEEE J. Solid-State Circuits*, vol. 42, pp. 219–232, Jan. 2007.

[41] K. Takeuchi, T. Tanaka, and H. Nakamura, "A Double-level-Vth Select Gate Array Architecture for Multilevel NAND Flash Memories," *IEEE J. Solid-State Circuits*, vol. 31, no. 4, pp. 602–609, Apr. 1996.

[42] D. Wellekens, J. Van Houdt, L. Faraone, G. Groeseneken, and H. Maes, "Write/erase Degradation in Source Side Injection Flash EEPROM's: Characterization Techniques and Wearout Mechanisms," *IEEE Transactions on Electron Devices*, vol. 42, no. 11, pp. 1992–1998, 1995.

[43] B. L. Worthington, G. R. Ganger, and Y. N. Patt, "Scheduling Algorithms for Modern Disk Drives," in *Proc. of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 1994, pp. 241–251.

[44] M. Wu and W. Zwaenepoel, “eNVy: a NonVolatile Main Memory Storage System,” *Proc. of Fourth Workshop on Workstation Operating Systems*, pp. 116 –118, Oct. 1993.

[45] H. Yang and et al., “Reliability Issues and Models of sub-90nm NAND Flash Memory Cells,” in *Proc. of International Conference on Solid-State and Integrated Circuit Technology*, 2006, pp. 760–762.

FAST: Quick Application Launch on Solid-State Drives

Yongsoo Joo†, Junhee Ryu‡, Sangsoo Park†, and Kang G. Shin†*
†Ewha Womans University, 11-1 Daehyun-dong Seodaemun-gu, Seoul 120-750, Korea
‡Seoul National University, 599 Kwanak-Gu Kwanak Rd., Seoul 151-744, Korea
* University of Michigan, 2260 Hayward St., Ann Arbor, MI 48109, USA

Abstract

Application launch performance is of great importance to system platform developers and vendors as it greatly affects the degree of users' satisfaction. The single most effective way to improve application launch performance is to replace a hard disk drive (HDD) with a solid state drive (SSD), which has recently become affordable and popular. A natural question is then whether or not to replace the traditional HDD-aware application launchers with a new SSD-aware optimizer.

We address this question by analyzing the inefficiency of the HDD-aware application launchers on SSDs and then proposing a new SSD-aware application prefetching scheme, called the *Fast Application STarter* (FAST). The key idea of FAST is to overlap the computation (CPU) time with the SSD access (I/O) time during an application launch. FAST is composed of a set of user-level components and system debugging tools provided by the Linux OS (operating system). In addition, FAST uses a system-call wrapper to automatically detect application launches. Hence, FAST can be easily deployed in any recent Linux versions without kernel recompilation. We implemented FAST on a desktop PC with a SSD running Linux 2.6.32 OS and evaluated it by launching a set of widely-used applications, demonstrating an average of 28% reduction of application launch time as compared to PC without a prefetcher.

1 Introduction

Application launch performance is one of the important metrics for the design or selection of a desktop or a laptop PC as it critically affects the user-perceived performance. Unfortunately, application launch performance has not kept up with the remarkable progress of CPU performance that has thus far evolved according to Moore's law. As frequently-used or popular applications get "heavier" (by adding new functions) with each new release, their launch takes longer even if a new, powerful machine equipped with high-speed multi-core CPUs and several GBs of main memory is used. This undesirable trend is known to stem from the poor random access performance of hard disk drives (HDDs). When an application stored in a HDD is launched, up to thousands of block requests are sent to the HDD, and a significant portion of its launch time is spent on moving the disk head to proper track and sector positions, i.e., *seek* and *rotational* latencies. Unfortunately, the HDD seek and rotational latencies have not been improved much over the last few decades, especially compared to the CPU speed improvement. In spite of the various optimizations proposed to improve the HDD performance in launching applications, users must often wait tens of seconds for the completion of launching frequently-used applications, such as Windows Outlook.

A quick and easy solution to eliminate the HDD's seek and rotational latencies during an application launch is to replace the HDD with a solid state drive (SSD). A SSD consists of a number of NAND flash memory modules, and does not use any mechanical parts, unlike disk heads and arms of a conventional HDD. While the HDD access latency—which is the sum of seek and rotational latencies—ranges up to a few tens of milliseconds (ms), depending on the seek distance, the SSD shows a rather uniform access latency of about a few hundred microseconds (*u*s). Replacing a HDD with a SSD is, therefore, the single most effective way to improve application launch performance.

Until recently, using SSDs as the secondary storage of desktops or laptops has not been an option for most users due to the high cost-per-bit of NAND flash memories. However, the rapid advance of semiconductor technology has continuously driven the SSD price down, and at the end of 2009, the price of an 80 GB SSD has fallen below 300 US dollars. Furthermore, SSDs can be installed in existing systems without additional hardware or software support because they are usually equipped with the same interface as HDDs, and OSes see a SSD as a block device just like a HDD. Thus, end-users begin to use a SSD as their system disk to install the OS image and applications.

Although a SSD can significantly reduce the application launch time, it does not give users ultimate satisfaction for all applications. For example, using a SSD reduces the launch time of a heavy application from tens of seconds to several seconds. However, users will soon become used to the SSD launch performance, and will then want the launch time to be reduced further, just as they see from light applications. Furthermore, users will keep on adding functions to applications, making them heavier with each release and their launch time greater. According to a recent report [24], the growth of software is rapid and limited only by the ability of hardware. These call for the need to further improve application launch performance on SSDs.

Unfortunately, most previous optimizers for application launch performance are intended for HDDs and have not accounted for the SSD characteristics. Furthermore, some of them may rather be detrimental to SSDs. For example, running a disk defragmentation tool on a SSD is not beneficial at all because changing the physical location of data in the SSD does not affect its access latency. Rather, it generates unnecessary write and erase operations, thus shortening the SSD's lifetime.

In view of these, the first step toward SSD-aware optimization may be to simply disable the traditional optimizers designed for HDDs. For example, Windows 7 disables many functions, such as disk defragmentation, application prefetch, Superfetch, and Readyboost when it detects a SSD being used as a system disk [27]. Let's consider another example. Linux is equipped with four disk I/O schedulers: NOOP, anticipatory, deadline, and completely fair queueing. The NOOP scheduler almost does nothing to improve HDD access performance, thus providing the worst performance on a HDD. Surprisingly, it has been reported that NOOP shows better performance than the other three sophisticated schedulers on a SSD [11].

To the best of our knowledge, this is the first attempt to focus entirely on improving application launch performance on SSDs. Specifically, we propose a new application prefetching method, called the *Fast Application STarter* (FAST), to improve application launch time on SSDs. The key idea of FAST is to overlap the computation (CPU) time with the SSD access (I/O) time during each application launch. To achieve this, we monitor the sequence of block requests in each application, and launch the application simultaneously with a prefetcher that generates I/O requests according to the *a priori* monitored application's I/O request sequence. FAST consists of a set of user-level components, a system-call wrapper, and system debugging tools provided by the Linux OS. FAST can be easily deployed in most recent Linux versions without kernel recompilation. We have implemented and evaluated FAST on a desktop PC with a SSD running Linux 2.6.32, demonstrating an average of 28% reduction of application launch time as compared to PC without a prefetcher.

This paper makes the following contributions:

- Qualitative and quantitative evaluation of the inefficiency of traditional HDD-aware application launch optimizers on SSDs;
- Development of a new SSD-aware application prefetching scheme, called FAST; and
- Implementation and evaluation of FAST, demonstrating its superiority and deployability.

While FAST can be also applied to HDDs, its performance improvements are only limited to high I/O requirements of application launches on HDDs. We observed that existing application prefetchers outperformed FAST on HDDs by effectively optimizing disk head movements, which will be discussed further in Section 5.

The paper is organized as follows. In Section 2, we review other related efforts and discuss their performance in optimizing application launch on SSDs. Section 3 describes the key idea of FAST and presents an upper bound for its performance. Section 4 details the implementation of FAST on the Linux OS, while Section 5 evaluates its performance using various real-world applications. Section 6 discusses the applicability of FAST to smartphones and Section 7 compares FAST with traditional I/O prefetching techniques. We conclude the paper with Section 8.

2 Background

2.1 Application Launch Optimization

Application-level optimization. Application developers are usually advised to optimize their applications for fast startup. For example, they may be advised to postpone loading non-critical functions or libraries so as to make applications respond as fast as possible [2, 30]. They are also advised to reduce the number of symbol relocations while loading libraries, and to use dynamic library loading. There have been numerous case studies—based on in-depth analyses and manual optimizations—of various target applications/platforms, such as Linux desktop suite platform [8], a digital TV [17], and a digital still camera [33]. However, such an approach requires the experts' manual optimizations for each and every application. Hence, it is economically infeasible for general-purpose systems with many (dynamic) application programs.

Snapshot technique. A snapshot boot technique has also been suggested for fast startup of embedded systems [19], which is different from the traditional hibernate shutdown function in that a snapshot of the main memory after booting an OS is captured only once, and used repeatedly for every subsequent booting of the system. However, applying this approach for application launch is not practical for the following reasons. First, the page cache in main memory is shared by all applications, and separating only the portion of the cache content that is related to a certain application is not possible without extensive modification of the page cache. Furthermore, once an application is updated, its snapshot should be invalidated immediately, which incurs runtime overhead.

Prediction-based prefetch. Modern desktops are equipped with large (up to several GBs) main memory, and often have abundant free space available in the main memory. Prediction-based prefetching, such as Superfetch [28] and Preload [12], loads an application's code blocks in the free space even if the user does not explicitly express his intent to execute that particular application. These techniques monitor and analyze the users' access patterns to predict which applications to be launched in future. Consequently, the improvement of launch performance depends strongly on prediction accuracy.

Sorted prefetch. The Windows OS is equipped with an application prefetcher [36] that prefetches application code blocks in a sorted order of their logical block addresses (LBAs) to minimize disk head movements. A similar idea has also been implemented for Linux OS [15, 25]. We call these approaches *sorted prefetch*. It monitors HDD activities to maintain a list of blocks accessed during the launch of each application. Upon detection of an application launch, the application prefetcher immediately pauses its execution and begins to fetch the blocks in the list in an order sorted by their LBAs. The application launch is resumed after fetching all the blocks, and hence, no page miss occurs during the launch.

Application defragmentation. The block list information can also be used in a different way to further reduce the seek distance during an application launch. Modern OSes commonly support a HDD defragmentation tool that reorganizes the HDD layout so as to place each file in a contiguous disk space. In contrast, the defragmentation tool can relocate the blocks in the list of each application by their access order [36], which helps reduce the total HDD seek distance during the launch.

Data pinning on flash caches. Recently, flash cache has been introduced to exploit the advantage of SSDs at a cost comparable to HDDs. A flash cache can be integrated into traditional HDDs, which is called a *hybrid* HDD [37]. Also, a PCI card-type flash cache is available [26], which is connected to the mother board of a desktop or laptop PC. As neither seek nor rotational latency is incurred while accessing data in the flash cache, we can accelerate application launch by storing the code blocks of frequently-used applications, which is called a *pinned set*. Due to the small capacity of flash cache, how to determine the optimal pinned set subject to the capacity constraint is a key to making performance improvement, and a few results of addressing this problem have been reported [16, 18, 22]. We expect that FAST can be integrated with the flash cache for further improvement of performance, but leave it as part of our future work.

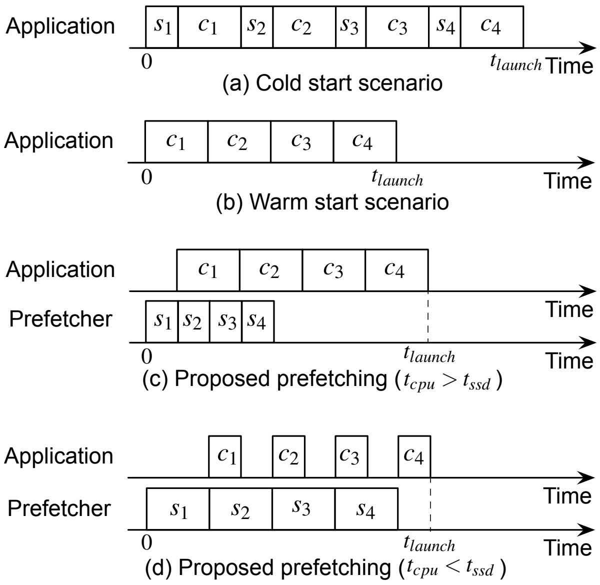
2.2 SSD Performance Optimization

SSDs have become affordable and begun to be deployed in desktop and laptop PCs, but their performance characteristics have not yet been understood well. So, researchers conducted in-depth analyses of their performance characteristics, and suggested ways to improve their runtime performance. Extensive experiments have been carried out to understand the performance dynamics of commercially-available SSDs under various workloads, without knowledge of their internal implementations [7]. Also, SSD design space has been explored and some guidelines to improve the SSD performance have been suggested [10]. A new write buffer management scheme has also been suggested to improve the random write performance of SSDs [20]. Traditional I/O schedulers optimized for HDDs have been revisited in order to evaluate their performance on SSDs, and then a new I/O scheduler optimized for SSDs has been proposed [11, 21].

2.3 Launch Optimization on SSDs

As discussed in Section 2.1, various approaches have been developed and deployed to improve the application launch performance on HDDs. On one hand, many of them are effective on SSDs as well, and orthogonal to FAST. For example, application-level optimization and prediction-based prefetch can be used together with FAST to further improve application launch performance.

On the other hand, some of them exploit the HDD characteristics to reduce the seek and rotational delay during an application launch, such as the sorted prefetch and the application defragmentation. Such methods are ineffective for SSDs because the internal structure of a SSD is very different from that of a HDD. A SSD typically consists of multiple NAND flash memory modules, and does not have any mechanical moving part. Hence, unlike a HDD, the access latency of a SSD is irrelevant to the LBA distance between the last and the current block



(a) Cold start scenario

(b) Warm start scenario

(c) Proposed prefetching ($t_{cpu} > t_{ssd}$)

(d) Proposed prefetching ($t_{cpu} < t_{ssd}$)

Figure 1: Various application launch scenarios ($n = 4$).

requests. Thus, prefetching the application code blocks according to the sorted order of their LBAs or changing their physical locations will not make any significant performance improvement on SSDs. As the sorted prefetch has the most similar structure to FAST, we will quantitatively compare its performance with FAST in Section 5.

3 Application Prefetching on SSDs

This section illustrates the main idea of FAST with examples and derives a lower bound of the application launch time achievable with FAST.

3.1 Cold and Warm Starts

We focus on the performance improvement in case of a cold start, or the first launch of an application upon system bootup, representing the worst-case application launch performance. Figure 1(a) shows an example cold start scenario, where s_i is the i-th block request generated during the launch and n the total number of block requests. After s_i is completed, the CPU proceeds with the launch process until another page miss takes place. Let c_i denote this computation.

The opposite extreme is a warm start in which all the code blocks necessary for launch have been found in the page cache, and thus, no block request is generated, as shown in Figure 1(b). This occurs when the application is launched again shortly after its closure. The warm start represents an upper-bound of the application launch performance improvement achievable with optimization of the secondary storage.

Let the time spent for s_i and c_i be denoted by $t(s_i)$ and $t(c_i)$, respectively. Then, the computation (CPU) time, t_{cpu}, is expressed as

$$t_{cpu} = \sum_{i=1}^{n} t(c_i), \qquad (1)$$

and the SSD access (I/O) time, t_{ssd}, is expressed as

$$t_{ssd} = \sum_{i=1}^{n} t(s_i). \qquad (2)$$

3.2 The Proposed Application Prefetcher

The rationale behind FAST is that the I/O request sequence generated during an application launch does not change over repeated launches of the application in case of cold-start. The key idea of FAST is to overlap the SSD access (I/O) time with the computation (CPU) time by running the application prefetcher concurrently with the application itself. The application prefetcher replays the I/O request sequence of the original application, which we call an *application launch sequence*. An application launch sequence S can be expressed as $(s_1, \ldots, s_n)$.

Figure 1(c) illustrates how FAST works, where $t_{cpu} > t_{ssd}$ is assumed. At the beginning, the target application and the prefetcher start simultaneously, and compete with each other to send their first block request to the SSD. However, the SSD always receives the same block request s_1 regardless of which process gets the bus grant first. After s_1 is fetched, the application can proceed with its launch by the time $t(c_1)$, while the prefetcher keeps issuing the subsequent block requests to the SSD. After completing c_1, the application accesses the code block corresponding to s_2, but no page miss occurs for s_2 because it has already been fetched by the prefetcher. It is the same for the remaining block requests, and thus, the resulting application launch time t_{launch} becomes

$$t_{launch} = t(s_1) + t_{cpu}. \qquad (3)$$

Figure 1(d) shows another possible scenario where $t_{cpu} < t_{ssd}$. In this case, the prefetcher cannot complete fetching s_2 before the application finishes computation c_1. However, s_2 can be fetched by $t(c_1)$ earlier than that of the cold start, and this improvement is accumulated for all of the remaining block requests, resulting in t_{launch}:

$$t_{launch} = t_{ssd} + t(c_n). \qquad (4)$$

Note that n ranges up to a few thousands for typical applications, and thus, $t(s_1) \ll t_{cpu}$ and $t(c_n) \ll t_{ssd}$. Consequently, Eqs. (3) and (4) can be combined into a single equation as:

$$t_{launch} \approx \max(t_{ssd}, t_{cpu}), \qquad (5)$$

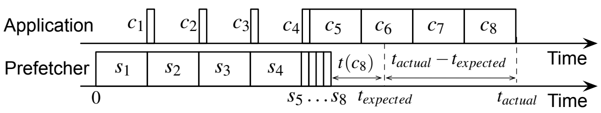


Figure 2: A worst-case example ($t_{cpu} = t_{ssd}$).

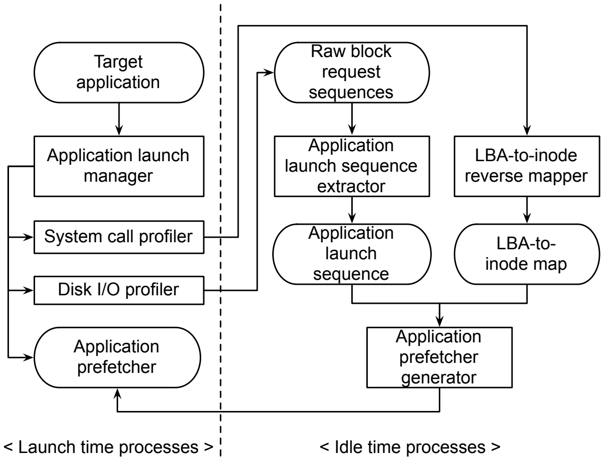

Target application
Application launch manager
System call profiler
Disk I/O profiler
Application prefetcher
Raw block request sequences
Application launch sequence extractor
Application launch sequence
LBA-to-inode reverse mapper
LBA-to-inode map
Application prefetcher generator
< Launch time processes >
< Idle time processes >

Figure 3: The proposed application prefetching.

which represents a lower bound of the application launch time achievable with FAST.

However, FAST may not achieve application launch performance close to Eq. (5) when there is a significant variation of I/O intensiveness, especially if the beginning of the launch process is more I/O intensive than the other. Figure 2 illustrates an extreme example of such a case, where the first half of this example is SSD-bound and the second half is CPU-bound. In this example, t_{cpu} is equal to t_{ssd}, and thus the expected launch time $t_{expected}$ is given to be $t_{ssd} + t(c_8)$, according to Eq. (4). However, the actual launch time t_{actual} is much larger than $t_{expected}$. The CPU usage in the first half of the launch time is kept quite low despite the fact that there are lots of remaining CPU computations (i.e., $c_5, \ldots, c_8$) due to the dependency between s_i and c_i. We will provide a detailed analysis for this case using real applications in Section 5.

4 Implementation

We chose the Linux OS to demonstrate the feasibility and the superior performance of FAST. The implementation of FAST consists of a set of components: an application launch manager, a system-call profiler, a disk I/O profiler, an application launch sequence extractor, a LBA-to-inode reverse mapper, and an application prefetcher generator. Figure 3 shows how these components interact with each other. In what follows, we detail the implementation of each of these components.

4.1 Application Launch Sequence

4.1.1 Disk I/O Profiler

The disk I/O profiler is used to track the block requests generated during an application launch. We used `Blktrace` [3], a built-in Linux kernel I/O-tracing tool that monitors the details of I/O behavior for the evaluation of I/O performance. `Blktrace` can profile various I/O events: inserting an item into the block layer, merging the item with a previous request in the queue, remapping onto another device, issuing a request to the device driver, and a completion signal from the device. From these events, we collect the trace of device-completion events, each of which consists of a device number, a LBA, the I/O size, and completion time.

4.1.2 Application Launch Sequence Extractor

Ideally, the application launch sequence should include all of the block requests that are generated every time the application is launched in the cold start scenario, without including any block requests that are not relevant to the application launch. We observed that the raw block request sequence captured by `Blktrace` does not vary from one launch to another, i.e., deterministic for multiple launches of the same application. However, we observed that other processes (e.g., OS and application daemons) sometimes generate their own I/O requests simultaneously with the application launch. To handle this case, the application launch sequence extractor collects two or more raw block request sequences to extract a common sequence, which is then used as a launch sequence of the corresponding application. The implementation of the application launch sequence extractor is simple: it searches for and removes any block requests appearing in some of the input sequences. This procedure makes all the input sequences the same, so we use any of them as an application launch sequence.

4.2 LBA-to-Inode Map

4.2.1 LBA-to-Inode Reverse Mapper

Our goal is to create an application prefetcher that generates exactly the same block request sequence as the obtained application launch sequence, where each block request is represented as a tuple of starting LBA and size. Since the application prefetcher is implemented as a user-level program, every disk access should be made via system calls with a file name and an offset in that file. Hence, we must obtain the file name and the offset of each block request in an application launch sequence.

Most file systems, including EXT3, do not support such a reverse mapping from LBA to file name and offset. However, for a given file name, we can easily find the LBA of all of the blocks that belong to the file and their relative offset in the file. Hence, we can build a LBA-to-inode map by gathering this information for every file. However, building such a map of the entire file system is time-consuming and impractical because a file system, in general, contains tens of thousands of files and their block locations on the disk change very often.

Therefore, we build a separate LBA-to-inode map for each application, which can significantly reduce the overhead of creating a LBA-to-inode map because (1) the number of applications and the number of files used in launching each application are very small compared to the number of files in the entire file system; and (2) most of them are shared libraries and application code blocks, so their block locations remain unchanged unless they are updated or disk defragmentation is performed.

We implement the LBA-to-inode reverse mapper that receives a list of file names as input and creates a LBA-to-inode map as output. A LBA-to-inode map is built using a red-black tree in order to reduce the search time. Each node in the red-black tree has the LBA of a block as its key, and a block type as its data by default. According to the block type, different types of data are added to the node. A block type includes a super block, a group descriptor, an inode block bitmap, a data block bitmap, an inode table, and a data block. For example, a node for a data block has a block type, a device number, an inode number, an offset, and a size. Also, for a data block, a table is created to keep the mapping information between an inode number and its file name.

4.2.2 System-Call Profiler

The system-call profiler obtains a full list of file names that are accessed during an application launch,[1] and passes it to the LBA-to-inode reverse mapper. We used `strace` for the system-call profiler, which is a debugging tool in Linux. We can specify the argument of `strace` so that it may monitor only the system calls that have a file name as their argument. As many of these system calls are rarely called during an application launch, we monitor only the following system calls that frequently occur during application launches: `open()`, `creat()`, `execve()`, `stat()`, `stat64()`, `lstat()`, `lstat64()`, `access()`, `truncate()`, `truncate64()`, `statfs()`, `statfs64()`, `readlink()`, and `unlink()`.

4.3 Application Prefetcher

4.3.1 Application Prefetcher Generator

The application prefetcher is a user-level program that replays the disk access requests made by a target application. We implemented the application prefetcher generator to automatically create an application prefetcher for each target application. It performs the following operations.

1. Read s_i one-by-one from S of the target application.
2. Convert s_i into its associated data items stored in the LBA-to-inode map, e.g.,
   `(dev,LBA,size)`→`(datablk,filename,offset,size)` or
   `(dev,LBA,size)`→`(inode,start_inode,end_inode)`.
3. Depending on the type of block, generate an appropriate system call using the converted disk access information.
4. Repeat Steps 1–3 until processing all s_i.

Table 1: System calls to replay access of blocks in an application launch sequence

Block type	System call
Inode table	`open()`
Data block: a directory	`opendir()` and `readdir()`
Data block: a regular file	`read()` or `posix_fadvise()`
Data block: a symbolic link file	`readlink()`

Table 1 shows the kind of system calls used for each block type. There are two system calls that can be used to replay the disk access for data blocks of a regular file. If we use `read()`, data is first moved from the SSD to the page cache, and then copying takes place from the page cache to the user buffer. The second step is unnecessary for our purpose, as the process that actually manipulates the data is not the application prefetcher but the target application. Hence, we chose `posix_fadvise()` that performs only the first step, from which we can avoid the overhead of `read()`. We use the `POSIX_FADV_WILLNEED` parameter, which informs the OS that the specified data will be used in the near future. When to issue the corresponding disk access after `posix_fadvise()` is called depends on the OS implementation. We confirmed that the current version of Linux we used issues a block request immediately after receiving the information through `posix_fadvise()`, thus meeting our need. A symbolic-linked file name is stored in data block pointers in an inode entry when the length of the file name is less than or equal to 60 bytes (c.f., the space of data block pointers is 60 bytes, 4*12 for direct, 4 for single indirect, another 4 for double indirect, and last 4 for triple indirect data block pointer). If the length of linked file name is more than 60 bytes, the name is stored in the data blocks pointed to by data block pointers in the inode entry. We use `readlink()` to replay the data block access of symbolic-link file names that are longer than 60 bytes.

[1] Files mounted on pseudo file systems such as `procfs` and `sysfs` are not processed because they never generate any disk I/O request.

```
int main(void) {
 ...
 readlink("/etc/fonts/conf.d/90-ttf-arphic-uming-emb
olden.conf", linkbuf, 256);
 int fd423;
 fd423 = open("/etc/fonts/conf.d/90-ttf-arphic-uming
-embolden.conf", O_RDONLY);
 posix_fadvise(fd423, 0, 4096, POSIX_FADV_WILLNEED);
 posix_fadvise(fd351, 286720, 114688, POSIX_FADV_WIL
LNEED);
 int fd424;
 fd424 = open("/usr/share/fontconfig/conf.avail/90-tt
f-arphic-uming-embolden.conf", O_RDONLY);
 posix_fadvise(fd424, 0, 4096, POSIX_FADV_WILLNEED);
 int fd425;
 fd425 = open("/root/.gnupg/trustdb.gpg", O_RDONLY);
 posix_fadvise(fd425, 0, 4096, POSIX_FADV_WILLNEED);
 dirp = opendir("/var/cache/");
 if(dirp)while(readdir(dirp));
 ...
 return 0;
}
```

Figure 4: An example application prefetcher.

Figure 4 is an example of automatically-generated application prefetcher. Unlike the target application, the application prefetcher successively fetches all the blocks as soon as possible to minimize the time between adjacent block requests.

4.3.2 Implicitly-Prefetched Blocks

In the EXT3 file system, the inode of a file includes pointers of up to 12 data blocks, so these blocks can be found immediately after accessing the inode. If the file size exceeds 12 blocks, indirect, double indirect, and triple indirect pointer blocks are used to store the pointers to the data blocks. Therefore, requests for indirect pointer blocks may occur in the cold start scenario when the application is accessing files larger than 12 blocks. We cannot explicitly load those indirect pointer blocks in the application prefetcher because there is no such system call. However, the `posix_fadvise()` call for a data block will first make a request for the indirect block when needed, so it can be fetched in a timely manner by running the application prefetcher.

The following types of block request are not listed in Table 1: a superblock, a group descriptor, an inode entry bitmap, a data block bitmap. We found that requests to these types of blocks seldom occur during an application launch, so we did not consider their prefetching.

4.4 Application Launch Manager

The role of the application launch manager is to detect the launch of an application and to take an appropriate action. We can detect the beginning of an application launch by monitoring `execve()` system call, which is implemented using a system-call wrapper. There are three phases with which the application launch manager deals: a launch profiling phase, a prefetcher generation phase, and an application prefetch phase. The application launch manager uses a set of variables and parameters for each application to decide when to change its phase. These are summarized in Table 2.

Table 2: Variables and parameters used by the application launch manager

Type	Description
n_{init}	A counter to record the number of application launches done in the initial launch phase
n_{pref}	A counter to record the number of launches done in the application prefetch phase after the last check of the miss ratio of the application prefetcher
N_{rawseq}	The number of raw block request sequences that are to be captured at the launch profiling phase
N_{chk}	The period to check the miss ratio of the application prefetcher
R_{miss}	A threshold value for the prefetcher miss ratio that is used to determine if an update of the application or shared libraries has taken place
T_{idle}	A threshold value for the idle time period that is used to determine if an application launch is completed
$T_{timeout}$	The maximum amount of time allowed for the disk I/O profiler to capture block requests

Here we describe the operations performed in each phase:

(1) Launch profiling. If no application prefetcher is found for that application, the application launch manager regards the current launch as the first launch of this application, and enters the initial launch phase. In this phase, the application launch manager performs the following operations in addition to the launch of the target application:

1. Increase n_{init} of the current application by 1.
2. If $n_{init} = 1$, run the system call profiler.
3. Flush the page cache, dentries (directory entries), and inodes in the main memory to ensure a cold start scenario, which is done by the following command:
   ```
   echo 3 > /proc/sys/vm/drop_caches
   ```
4. Run the disk I/O profiler. Terminate the disk I/O profiler when any of the following conditions are met: (1) if no block request occurs during the last T_{idle} seconds or (2) the elapsed time since the start of the disk I/O profiler exceeds $T_{timeout}$ seconds.
5. If $n_{init} = N_{rawseq}$, enter the prefetcher generation phase after the current launch is completed.

(2) Prefetcher generation. Once application launch profiling is done, it is ready to generate an application prefetcher using the information obtained from the first phase. This can be performed either immediately after the application launch is completed, or when the system is idle. The following operations are performed:

1. Run the application launch sequence extractor.
2. Run the LBA-to-inode reverse mapper.
3. Run the application prefetcher generator.
4. Reset the values of n_{init} and n_{pref} to 0.

(3) Application prefetch. If the application prefetcher for the current application is found, the application launch manager runs the prefetcher simultaneously with the target application. It also periodically checks the miss ratio of the prefetcher to determine if there has been any update of the application or shared libraries. Specifically, the following operations are performed:

1. Increase n_{pref} of the current application by 1.
2. If $n_{pref} = N_{chk}$, reset the value of n_{pref} to 0 and run the disk I/O profiler. Its termination conditions are the same as those in the first phase.
3. Run the application prefetcher simultaneously with the target application.
4. If a raw block request sequence is captured, use it to calculate the miss ratio of the application prefetcher. If it exceeds R_{miss}, delete the application prefetcher.

The miss ratio is defined as the ratio of the number of block requests not issued by the prefetcher to the total number of block requests in the application launch sequence.

5 Performance Evaluation

5.1 Experimental Setup

Experimental platform. We used a desktop PC equipped with an Intel i7-860 2.8 GHz CPU, 4GB of PC12800 DDR3 SDRAM and an Intel 80GB SSD (X25-M G2 Mainstream). We installed a Fedora 12 with Linux kernel 2.6.32 on the desktop, in which we set NOOP as the default I/O scheduler. For benchmark applications, we chose frequently used user-interactive applications, for which application launch performance matters much. Such an application typically uses graphical user interfaces and requires user interaction immediately after completing its launch. Applications like gcc and gzip are not included in our set of benchmarks as launch performance is not an issue for them. Our benchmark set consists of the following Linux applications: Acrobat reader, Designer-qt4, Eclipse, F-Spot, Firefox, Gimp, Gnome, Houdini, Kdevdesigner, Kdevelop, Konqueror, Labview, Matlab, OpenOffice, Skype, Thunderbird, and XilinxISE. In addition to these, we used Wine [1], which is an implementation of the Windows API running on the Linux OS, to test Access, Excel, Powerpoint, Visio, and Word—typical Windows applications.

Test scenarios. For each benchmark application, we measured its launch time for the following scenarios.

- *Cold start*: The application is launched immediately after flushing the page cache, using the method described in Section 4.4. The resulting launch time is denoted by t_{cold}.
- *Warm start*: We first run the application prefetcher only to load all the blocks in the application launch sequence to the page cache, and then launch the application. Let t_{warm} denote the resulting launch time.
- *Sorted prefetch*: To evaluate the performance of the sorted prefetch [15, 25, 36] on SSDs, we modify the application prefetcher to fetch the block requests in the application launch sequence in the sorted order of their LBAs. After flushing the page cache, we first run the modified application prefetcher, then immediately run the application. Let t_{sorted} denote the resulting launch time.
- *FAST*: We flush the page cache, and then run the application simultaneously with the application prefetcher. The resulting launch time is denoted by t_{FAST}.
- *Prefetcher only*: We flush the page cache and run the application prefetcher. The completion time of the application prefetcher is denoted by t_{ssd}. It is used to calculate a lower bound of the application launch time $t_{bound} = \max(t_{ssd}, t_{cpu})$, where $t_{cpu} = t_{warm}$ is assumed.

Launch-time measurement. We start an application launch by clicking an icon or inputting a command, and can accurately measure the launch start time by monitoring when `execve()` is called. Although it is difficult to clearly define the completion of a launch, a reasonable definition is the first moment the application becomes responsive to the user [2]. However, it is difficult to accurately and automatically measure that moment. So, as an alternative, we measured the completion time of the last block request in an application launch sequence using `Blktrace`, assuming that the launch will be completed very soon after issuing the last block request. For the warm start scenario, we executed `posix_fadvise()` with `POSIX_FADV_DONTNEED` parameter to evict the last block request from the page cache. For the sorted prefetch and the FAST scenarios, we modified the application prefetcher so that it skips prefetching of the last block request.

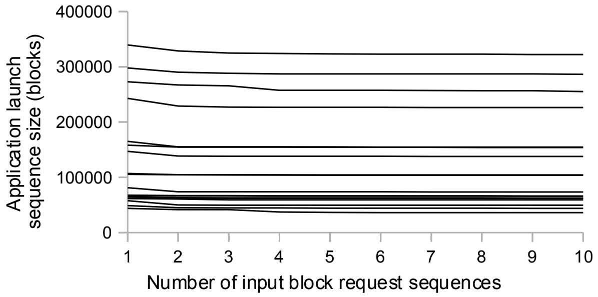


Figure 5: The size of application launch sequences.

5.2 Experimental Results

Application launch sequence generation. We captured 10 raw block request sequences during the cold start launch of each application. We ran the application launch sequence extractor with a various number of input block request sequences, and observed the size of the resulting application launch sequences. Figure 5 shows that for all the applications we tested, there is no significant reduction of the application launch sequence size while increasing the number of inputs from 2 to 10. Hence, we set the value of N_{rawseq} in Table 2 to 2 in this paper. We used the size of the first captured input sequence as the number of inputs one in Figure 5 (the application launch sequence extractor requires at least two input sequences). For some applications, there are noticeable differences in size between the number of inputs one and two. This is because the first raw input request sequence includes a set of bursty I/O requests generated by OS and user daemons that are irrelevant to the application launch. Figure 5 shows that such I/O requests can be effectively excluded from the resulting application launch sequence using just two input request sequences.

The second and third columns of Table 3 summarize the total number of block requests and accessed blocks of the thus-obtained application launch sequences, respectively. The last column shows the total number of files used during the launch of each application.

Testing of the application prefetcher. Application prefetchers are automatically generated for the benchmark applications using the application launch sequences in Table 3. In order to see if the application prefetchers fetch all the blocks used by an application, we first flushed the page cache, and launched each application immediately after running the application prefetcher. During the application launch, we captured all the block requests generated using `Blktrace`, and counted the number of missed block requests. The average number of missed block requests was 1.6% of the number of block requests in the application launch sequence, but varied among repeated launches, e.g., from 0% to 6.1% in the experiments we performed.

Table 3: Collected launch sequences ($N_{rawseq} = 2$)

Application	# of block requests	# of fetched blocks	# of used files
Access	1296	106 992	555
Acrobat reader	960	73 784	178
Designer-qt4	2400	138 608	410
Eclipse	4163	155 216	787
Excel	1610	169 112	583
F-Spot	1180	49 968	304
Firefox	1566	60 944	433
Gimp	1939	66 928	799
Gnome	4739	228 872	538
Houdini	4836	290 320	724
Kdevdesigner	1537	44 904	467
Kdevelop	1970	63 104	372
Konqueror	1780	62 216	296
Labview	2927	154 768	354
Matlab	6125	267 312	742
OpenOffice	1425	104 600	308
Powerpoint	1405	120 808	576
Skype	892	41 560	197
Thunderbird	1533	64 784	429
Visio	1769	168 832	662
Word	1715	181 496	613
Xilinx ISE	4718	328 768	351

By examining the missed block requests, we could categorize them into three types: (1) files opened by OS daemons and user daemons at boot time; (2) journaling data or swap partition accesses; and (3) files dynamically created or renamed at every launch (e.g., `tmpfile()`). The first type occurs because we force the page cache to be flushed in the experiment. In reality, they are highly likely to reside in the page cache, and thus, this type of misses will not be a problem. The second type is irrelevant to the application, and observed even during idle time. The third type occurs more or less often, depending on the application. FAST does not prefetch this type of block requests as they change at every launch.

Experiments for the test scenarios. We measured the launch time of the benchmark applications for each test scenario listed in Section 5.1. Figure 6 shows that the average launch time reduction of FAST is 28% over the cold start scenario. The performance of FAST varies considerably among applications, ranging from 16% to 46% reduction of launch time. In particular, FAST shows performance very close to t_{bound} for some applications, such as Eclipse, Gnome, and Houdini. On the other hand, the gap between t_{bound} and t_{FAST} is relatively larger for such applications as Acrobat reader, Firefox, OpenOffice, and Labview.

Launch time behavior. We conducted experiments to see if the application prefetcher works well as expected when it is simultaneously run with the application. We

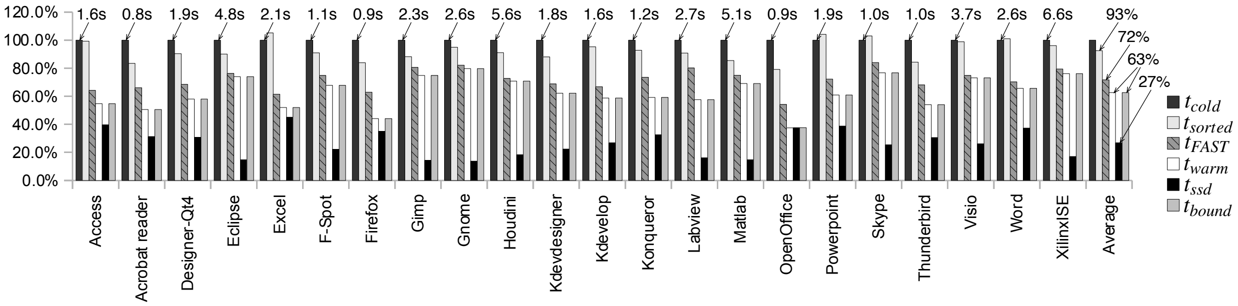


Figure 6: Measured application launch time (normalized to t_{cold}).

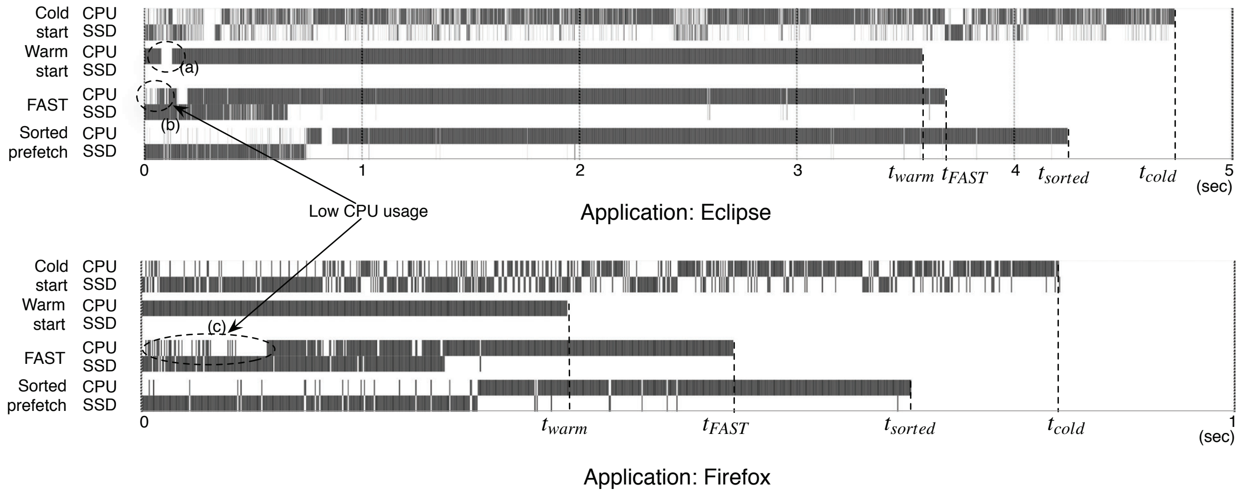


Figure 7: Usage of CPU and SSD (sampling rate = 1 KHz).

chose Firefox because it shows a large gap between t_{bound} and t_{FAST}. We monitored the generated block requests during the launch of Firefox with the application prefetcher, and observed that the first 12 of the entire 1566 block requests were issued by Firefox, which took about 15 ms. As the application prefetcher itself should be launched as well, FAST cannot prefetch these block requests until finishing its launch. However, we observed that all the remaining block requests were issued by FAST, meaning that they are successfully prefetched before the CPU needs them.

CPU and SSD usage patterns. We performed another experiment to observe the CPU and SSD usage patterns in each test scenario. We chose two applications, Eclipse and Firefox, representing the two groups of applications of which t_{FAST} is close to and far from t_{bound}, respectively. We modified the OS kernel to sample the number of CPU cores having runnable processes and to count the number of cores in the I/O wait state. Figure 7 shows the CPU and SSD usage of the two applications, where the entire CPU is regarded as busy if at least one of its cores is active. Similarly, the SSD is assumed busy if there are one or more cores in the I/O wait state. In the cold start scenario, there is almost no overlap between CPU computation and SSD access for both applications. In the warm start scenario, the CPU stays fully active until the launch is completed as there is no wait. One exception we observed is the time period marked with Circle (a), during which the CPU seems to be in the event-waiting state. FAST is shown to be successful in overlapping CPU computation with SSD access as we intended. However, CPU usage is observed to be low at the beginning of launch for both applications, which can be explained with the example in Figure 2. As Eclipse shows a shorter such time period (Circle (b)) than Firefox (Circle (c)), t_{FAST} can reach closer to t_{bound}. In the case of Firefox, however, the ratio of t_{cpu} to t_{ssd} is close to 1:1, allowing FAST to achieve more reduction of launch time for Firefox than for Eclipse.

Performance of sorted prefetch. Figure 6 shows that the sorted prefetch reduces the application launch time by an average of 7%, which is less efficient than FAST, but non-negligible. One reason for this improvement is the difference in I/O burstiness between the cold start and the sorted prefetch. Most SSDs (including the one we used) support the native command queueing (NCQ)

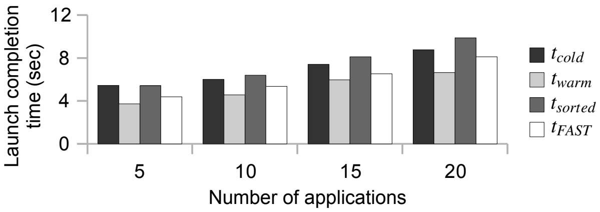


Figure 8: Simultaneous launch of multiple applications.

feature, which allows up to 31 block requests to be sent to a SSD controller. Using this information, the SSD controller can read as many NAND flash chips as possible, effectively increasing read throughput. The average queue depth in the cold start scenario is close to 1, meaning that for most of time there is only one outstanding request in case of SSD. In contrast, in the sorted prefetch scenario, the queue depth will likely grow larger than 1 because the prefetcher may successively issue asynchronous I/O requests using `posix_fadvise()`, at small inter-issue intervals.

On the other hand, we could not find a clear evidence that sorting block requests in their LBA order is advantageous in case of SSD. Rather, the execution time of the sorted prefetcher was slightly longer than its unsorted version for most of the applications we tested. Also, the sorted prefetch shows worse performance than the cold start for Excel, Powerpoint, Skype, and Word. Although these observations were consistent over repeated tests, a further investigation is necessary to understand such a behavior.

Simultaneous launch of applications. We performed experiments to see how well FAST can scale up for launching multiple applications. We launched multiple applications starting from the top of Table 3, adding five at a time, and measured the launch completion time of all launched applications[2]. Figure 8 shows that FAST could reduce the launch completion time for all the tests, whereas the sorted prefetch does not scale beyond 10 applications. Note that the FAST improvement decreased from 20% to 7% as the number of applications increased from 5 to 20.

Table 4: Runtime overhead (application: Firefox)

Running processes	Runtime (sec)
1. Application only (cold start scenario)	0.86
2. `strace` + `blktrace` + application	1.21
3. `blktrace` + application	0.88
4. Prefetcher generation	5.01
5. Prefetcher + application	0.56
6. Prefetcher + `blktrace` + application	0.59
7. Miss ratio calculation	0.90

Runtime and space overhead. We analyzed the runtime overhead of FAST for seven possible combinations of running processes, and summarized the results in Table 4. Cases 2 and 3 belong to the *launch profiling* phase, which was described in Section 4.4. During this phase, Case 2 occurs only once, and Case 3 occurs N_{rawseq} times. Case 4 corresponds to the *prefetcher generation* phase (the right side of Figure 3), and shows a relatively long runtime. However, we can hide it from users by running it in background. Also, since we primarily focused on functionality in the current implementation, there is room for further optimization. Cases 5, 6, and 7 belong to the *application prefetch* phase, and repeatedly occur until the application prefetcher is invalidated. Cases 6 and 7 occur only when n_{pref} reaches N_{chk}, and Case 7 can be run in background.

FAST creates temporary files such as system call log files and I/O traces, but these can be deleted after FAST completes creating application prefetchers. However, the generated prefetchers occupy disk space as far as application prefetching is used. In addition, application launch sequences are stored to check the miss ratio of the corresponding application prefetcher. In our experiment, the total size of the application prefetchers and application launch sequences for all 22 applications was 7.2 MB.

FAST applicability. While previous examples clearly demonstrated the benefits of FAST for a wide range of applications, FAST does not guarantee improvements for all cases. One such a scenario is when a target application is too small to offset the overhead of loading the prefetcher. We tested FAST with the Linux utility `uname`, which displays the name of the OS. It generated 3 I/O requests whose total size was 32 KB. The measured t_{cold} was 2.2 ms, and t_{FAST} was 2.3 ms, 5% longer than the cold start time.

Another possible scenario is when the target application experiences a major update. In this scenario, FAST may fetch data that will not be used by the newly updated application until it detects the application update and enters a new launch profiling phase. We modified the application prefetcher so that it fetches the same size of data from the same file but from another offset that is not used by the application. We tested the modified prefetcher with Firefox. Even in this case, FAST reduced application launch time by 4%, because FAST could still prefetch some of the metadata used by the application. Assuming most of the file names are changed after the update, we ran Firefox with the prefetcher for Gimp, which fetches a similar number of blocks as Firefox. In this experiment, the measured application launch time was 7% longer than the cold start time, but the performance degradation was not drastic due to the internal parallelism of the SSD we used (10 channels).

[2] Except for Gnome that cannot be launched with other applications, and Houdini whose license had expired.

Configuring application launch manager. The application launch manager has a set of parameters to be configured, as shown in Table 2. If N_{rawseq} is set too large, users will experience the cold-start performance during the initialization phase. If it is set too small, unnecessary blocks may be included in the application prefetcher. Figure 5 shows that setting it between 2 and 4 is a good choice. The proper value of N_{chk} will depend on the runtime overhead of `Blktrace`; if FAST is placed in the OS kernel, the miss ratio of the application prefetcher may be checked upon every launch ($N_{chk} = 1$) without noticeable overhead. Also, setting R_{miss} to 0.1 is reasonable, but it needs to be adjusted after gaining enough experience in using FAST. To find the proper value of T_{idle}, we investigated the SSD's maximum idle time during the cold-start of applications, and found it to range from 24 ms (Thunderbird) to 826 ms (Xilinx ISE). Hence, setting T_{idle} to 2 seconds is proper in practice. As the maximum cold-start launch time is observed to be less than 10 seconds, 30 seconds may be reasonable for $T_{timeout}$. All these values may need to be adjusted, depending on the underlying OS and applications.

Running FAST on HDDs. To see how FAST works on a HDD, we replaced the SSD with a Seagate 3.5" 1 TB HDD (ST31000528AS) and measured the launch time of the same set of benchmark applications. Although FAST worked well as expected by hiding most of CPU computation from the application launch, the average launch time reduction was only 16%. It is because the application launch on a HDD is mostly I/O bound; in the cold start scenario, we observed that about 85% of the application launch time was spent on accessing the HDD. In contrast, the sorted prefetch was shown to be more effective; it could reduce the application launch time by an average of 40% by optimizing disk head movements.

We performed another experiment by modifying the sorted prefetch so that the prefetcher starts simultaneously with the original application, like FAST. However, the resulting launch time reduction was only 19%, which is worse than that of the unmodified sorted prefetch. The performance degradation is due to the I/O contention between the prefetcher and the application.

6 Applicability of FAST to Smartphones

The similarity between modern smartphones and PCs with SSDs in terms of the internal structure and the usage pattern, as summarized below, makes smartphones a good candidate to which we can apply FAST:

- Unlike other mobile embedded systems, smartphones run different applications at different times, making application launch performance matter more;

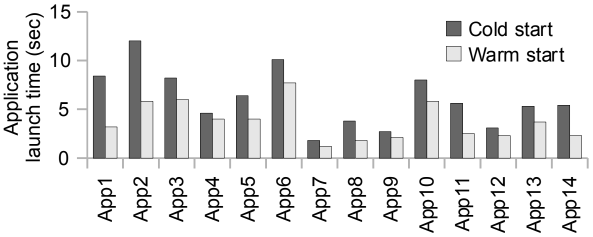


Figure 9: Measured application launch time on iPhone 4 (CPU: 1 GHz, SDRAM: 512 MB, NAND flash: 32 GB).

- Smartphones use NAND flash as their secondary storage, of which the performance characteristics are basically the same as the SSD; and
- Smartphones often use slightly customized (if not the same) OSes and file systems that are designed for PCs, reducing the effort to port FAST to smartphones.

Furthermore, a smartphone has the characteristics that enhance the benefit of using FAST as follows:

- Users tend to launch and quit applications more frequently on smartphones than on PCs;
- Due to relatively smaller main memory of a smartphone, users will experience cold start performance more frequently; and
- Its relatively slower CPU and flash storage speed may increase the absolute reduction of application launch time by applying FAST.

Although we have not yet implemented FAST on a smartphone, we could measure the launch time of some smartphone applications by simply using a stopwatch. We randomly chose 14 applications installed on the iPhone 4 to compare their cold and warm start times, of which the results are plotted in Figure 9. The average cold start time of the smartphone applications is 6.1 seconds, which is more than twice of the average cold start time of the PC applications (2.4 seconds) shown in Figure 6. Figure 9 also shows that the average warm start time is 63% of the cold start time (almost the same ratio as in Figure 6), implying that we can achieve similar benefits from applying FAST to smartphones.

7 Comparison of FAST with Traditional Prefetching

FAST is a special type of prefetching optimized for application launch, whereas most of the traditional prefetching schemes focus on runtime performance improvement. We compare FAST with the traditional prefetching algorithms by answering the following three questions that are inspired by previous work [32].

7.1 What to Prefetch

FAST prefetches the blocks appeared in the application launch sequence. While many prediction-based prefetching schemes [9, 23, 39] suffer from the low hit ratio of the prefetched data, FAST can achieve near 100% hit ratio. This is because the application launch sequence changes little over repeated launches of an application, as observed by previous work [4, 18, 34].

Sequential pattern detection schemes like readahead [13, 31] can achieve a fairly good hit ratio when activated, but they are applicable only when such a pattern is detected. By contrast, FAST guarantees stable performance improvement for every application launch.

One way to enhance the prefetch hit ratio for a complicated disk I/O pattern is to analyze the application source code to extract its access pattern. Using the thus-obtained pattern, prefetching can be done by either inserting prefetch codes into the application source code [29, 38] or converting the source code into a computation thread and a prefetch thread [40]. However, such an approach does not work well for application launch optimization because many of the block requests generated during an application launch are not from the application itself but from other sources, such as loading shared libraries, which cannot be analyzed by examining the application source code. Furthermore, both require modification of the source code, which is usually not available for most commercial applications. Even if the source code is available, modifying and recompiling every application would be very tedious and inconvenient. In contrast, FAST does not require application source code and is thus applicable for any commercial application.

Another relevant approach [6] is to deploy a shadow process that speculatively executes the copy of the original application to get hints for the future I/O requests. It does not require any source modification, but consumes non-negligible CPU and memory resources for the shadow process. Although it is acceptable when CPU is otherwise stalled waiting for the I/O completion, employing such a shadow process in FAST may degrade application launch performance as there is not enough CPU idle period as shown in Figure 7.

7.2 When to Prefetch

FAST is not activated until an application is launched, which is as conservative as demand paging. Thus, unlike prediction-based application prefetching schemes [12, 28], there is no cache-pollution problem or additional disk I/O activity during idle period. However, once activated, FAST aggressively performs prefetching: it keeps on fetching subsequent blocks in the application launch sequence *asynchronously* even in the absence of page misses. As the prefetched blocks are mostly (if not all) used by the application, the performance improvement of FAST is comparable to that of the prediction-based schemes when their prediction is accurate.

7.3 What to Replace

FAST does not modify the replacement algorithm of page cache in main memory, so the default page replacement algorithm is used to determine which page to evict in order to secure free space for the prefetched blocks.

In general, prefetching may significantly affect the performance of page replacement. Thus, previous work [5, 14, 35] emphasized the need for integrated prefetching and caching. However, FAST differs from the traditional prefetching schemes since it prefetches only those blocks that will be referenced before the application launch completes (e.g., in next few seconds). If the page cache in the main memory is large enough to store all the blocks in the application launch sequence, which is commonly the case, FAST will have minimal effect on the optimality of the page replacement algorithm.

8 Conclusion

We proposed a new I/O prefetching technique called *FAST* for the reduction of application launch time on SSDs. We implemented and evaluated FAST on the Linux OS, demonstrating its deployability and performance superiority. While the HDD-aware application launcher showed only 7% of launch time reduction on SSDs, FAST achieved a 28% reduction with no additional overhead, demonstrating the need for, and the utility of, a new SSD-aware optimizer. FAST with a well-designed entry-level SSD can provide end-users the fastest application launch performance. It also incurs fairly low implementation overhead and has excellent portability, facilitating its wide deployment in various platforms.

Acknowledgments

We deeply appreciate Prof. Heonshik Shin for his support and providing research facility. We also thank our shepherd Arkady Kanevsky, and the anonymous reviewers for their invaluable comments that improved this paper. This research was supported by WCU (World Class University) program through National Research Foundation of Korea funded by the Ministry of Education, Science and Technology (R33-10085), and RP-Grant 2010 of Ewha Womans University. Sangsoo Park is the corresponding author (email: sangsoo.park@ewha.ac.kr).

References

[1] *Wine User Guide*. http://www.winehq.org/docs/wineusr-guide/index, Last accessed on: 17 November 2010.

[2] APPLE INC. *Launch Time Performance Guidelines*. http://developer.apple.com/documentation/Performance/Conceptual/LaunchTime/LaunchTime.pdf, 2006.

[3] AXBOE, J. *Block IO Tracing*. http://www.kernel.org/git/?p=linux/kernel/git/axboe/blktrace.git;a=blob;f=README, 2006.

[4] BHADKAMKAR, M., GUERRA, J., USECHE, L., BURNETT, S., LIPTAK, J., RANGASWAMI, R., AND HRISTIDIS, V. BORG: Block-reORGanization for self-optimizing storage systems. In *Proc. FAST* (2009), pp. 183–196.

[5] CAO, P., FELTEN, E. W., KARLIN, A. R., AND LI, K. A study of integrated prefetching and caching strategies. In *Proc. SIGMETRICS* (1995), pp. 188–197.

[6] CHANG, F., AND GIBSON, G. A. Automatic I/O hint generation through speculative execution. In *Proc. OSDI* (1999), pp. 1–14.

[7] CHEN, F., KOUFATY, D. A., AND ZHANG, X. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proc. SIGMETRICS* (2009), pp. 181–192.

[8] COLITTI, L. Analyzing and improving GNOME startup time. In *Proc. SANE* (2006), pp. 1–11.

[9] CUREWITZ, K. M., KRISHNAN, P., AND VITTER, J. S. Practical prefetching via data compression. *SIGMOD Rec. 22*, 2 (1993), 257–266.

[10] DIRIK, C., AND JACOB, B. The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device architecture, and system organization. In *Proc. ISCA* (2009), pp. 279–289.

[11] DUNN, M., AND REDDY, A. L. N. A new I/O scheduler for solid state devices. Tech. Rep. TAMU-ECE-2009-02, Department of Electrical and Computer Engineering, Texas A&M University, 2009.

[12] ESFAHBOD, B. Preload—An adaptive prefetching daemon. Master's thesis, Graduate Department of Computer Science, University of Toronto, Canada, 2006.

[13] FENGGUANG, W., HONGSHENG, X., AND CHENFENG, X. On the design of a new Linux readahead framework. *SIGOPS Oper. Syst. Rev. 42*, 5 (2008), 75–84.

[14] GILL, B. S., AND MODHA, D. S. SARC: Sequential prefetching in adaptive replacement cache. In *Proc. USENIX* (2005), pp. 293–308.

[15] HUBERT, B. On faster application startup times: Cache stuffing, seek profiling, adaptive preloading. In *Proc. OLS* (2005), pp. 245–248.

[16] INTEL. *Intel Turbo Memory with User Pinning*. Intel, http://www.intel.com/design/flash/nand/turbomemory/index.htm, Last accessed on: 17 November 2010.

[17] JO, H., KIM, H., JEONG, J., LEE, J., AND MAENG, S. Optimizing the startup time of embedded systems: A case study of digital TV. *IEEE Trans. Consumer Electron. 55*, 4 (2009), 2242–2247.

[18] JOO, Y., CHO, Y., LEE, K., AND CHANG, N. Improving application launch times with hybrid disks. In *Proc. CODES+ISSS* (2009), pp. 373–382.

[19] KAMINAGA, H. Improving Linux startup time using software resume. In *Proc. OLS* (2006), pp. 25–34.

[20] KIM, H., AND AHN, S. BPLRU: A buffer management scheme for improving random writes in flash storage. In *Proc. FAST* (2008), pp. 1–14.

[21] KIM, J., OH, Y., KIM, E., CHOI, J., LEE, D., AND NOH, S. H. Disk schedulers for solid state drivers. In *Proc. EMSOFT* (2009), pp. 295–304.

[22] KIM, Y.-J., LEE, S.-J., ZHANG, K., AND KIM, J. I/O performance optimization techniques for hybrid hard disk-based mobile consumer devices. *IEEE Trans. Consumer Electron. 53*, 4 (2007), 1469–1476.

[23] KOTZ, D., AND ELLIS, C. S. Practical prefetching techniques for parallel file systems. In *Proc. PDIS* (1991), pp. 182–189.

[24] LARUS, J. Spending Moore's dividend. *Commun. ACM 52*, 5 (2009), 62–69.

[25] LICHOTA, K. *Prefetch: Linux solution for prefetching necessary data during application and system startup*. http://code.google.com/p/prefetch/, 2007.

[26] MATTHEWS, J., TRIKA, S., HENSGEN, D., COULSON, R., AND GRIMSRUD, K. Intel®Turbo Memory: Nonvolatile disk caches in the storage hierarchy of mainstream computer systems. *ACM Trans. Storage 4*, 2 (2008), 1–24.

[27] MICROSOFT. *Support and Q&A for Solid-State Drives*. Microsoft, http://blogs.msdn.com/e7/archive/2009/05/05/support-and-q-a-for-solid-state-drives-and.aspx, 2009.

[28] MICROSOFT. *Windows PC Accelerators*. http://www.microsoft.com/whdc/system/sysperf/perfaccel.mspx, Last accessed on: 17 November 2010.

[29] MOWRY, T. C., DEMKE, A. K., AND KRIEGER, O. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proc. OSDI* (1996), pp. 3–17.

[30] NEELAKANTH NADGIR. *Reducing Application Startup Time in the Solaris 8 OS*. http://developers.sun.com/solaris/articles/reducing_app.html, 2002.

[31] PAI, R., PULAVARTY, B., AND CAO, M. Linux 2.6 performance improvement through readahead optimization. In *Proc. OLS* (2004), pp. 105–116.

[32] PAPATHANASIOU, A. E., AND SCOTT, M. L. Energy efficient prefetching and caching. In *Proc. USENIX* (2004), pp. 22–22.

[33] PARK, C., KIM, K., JANG, Y., AND HYUN, K. Linux bootup time reduction for digital still camera. In *Proc. OLS* (2006), pp. 239–248.

[34] PARUSH, N., PELLEG, D., BEN-YEHUDA, M., AND TA-SHMA, P. Out-of-band detection of boot-sequence termination events. In *Proc. ICAC* (2009), pp. 71–72.

[35] PATTERSON, R. H., GIBSON, G. A., GINTING, E., STODOLSKY, D., AND ZELENKA, J. Informed prefetching and caching. In *Proc. SOSP* (1995), pp. 79–95.

[36] RUSSINOVICH, M. E., AND SOLOMON, D. *Microsoft Windows Internals*, 4th ed. Microsoft Press, 2004, pp. 458–462.

[37] SAMSUNG SEMICONDUCTOR. *Samsung Hybrid Hard Drive*. http://www.samsung.com/global/business/semiconductor/support/brochures/downloads/hdd/hd_datasheet_200708.pdf, 2007.

[38] VANDEBOGART, S., FROST, C., AND KOHLER, E. Reducing seek overhead with application-directed prefetching. In *Proc. USENIX* (2009).

[39] VELLANKI, V., AND CHERVENAK, A. L. A cost-benefit scheme for high performance predictive prefetching. In *Proc. SC* (1999).

[40] YANG, C.-K., MITRA, T., AND CHIUEH, T.-C. A decoupled architecture for application-specific file prefetching. In *Proc. USENIX* (2002), pp. 157–170.

Cost Effective Storage using Extent Based Dynamic Tiering

Jorge Guerra[†], Himabindu Pucha[*], Joseph Glider[*], Wendy Belluomini[*], Raju Rangaswami[†]
[†]*Florida International University, IBM Research Almaden*[*]

Abstract

Multi-tier systems that combine SSDs with SAS/FC and/or SATA disks mitigate the capital cost burden of SSDs, while benefiting from their superior I/O performance per unit cost and low power. Though commercial SSD-based multi-tier solutions are available, configuring such a system with the optimal number of devices per tier to achieve performance goals at minimum cost remains a challenge. Furthermore, these solutions do not leverage the opportunity to dynamically consolidate load and reduce power/operating cost.

Our extent-based dynamic tiering solution, *EDT*, addresses these limitations via two key components of its design. A Configuration Adviser *EDT-CA* determines the adequate mix of storage devices to buy and install to satisfy a given workload at minimum cost, and a Dynamic Tier Manager *EDT-DTM* performs dynamic extent placement once the system is running to satisfy performance requirements while minimizing dynamic power consumption. Key to the cost minimization of EDT-CA is its ability to simulate the dynamic extent placement afforded by EDT-DTM. Key to the overall effectiveness of EDT-DTM is its ability to consolidate load within tiers when feasible, rapidly respond to unexpected changes in the workload, and carefully control the overhead due to extent migration. Our results using production workloads show that EDT incurs lower capital and operating cost, consumes less power, and delivers similar or better performance relative to SAS-only storage systems as well as other simpler approaches to extent-based tiering.

1 Introduction

Enterprise storage systems strive to provide performance and reliability at minimum capital and operating cost. These systems use high performance disk drives (e.g. SCSI/SAS/FC) to provide that performance. However, solid-state drives (SSDs) offering superior random access capability per GByte have become increasingly affordable. On the other hand, SATA drives offering superior cost per GByte are also attractive for mass storage. Systems with only SSDs are still too expensive, and those built using only SATA would not provide enough performance/GByte for most enterprise workloads. *Multi-tier* systems containing a mix of devices can provide high performing and lower cost storage by utilizing SSDs only for the subset of the data that needs SSD performance.

Current commercial SSD-based multi-tier systems from IBM [29], EMC [17], 3PAR [25] and Compellent [23] provide performance gains and cost savings. However, customer adoption has been slow. One of the reasons for this is the difficulty in determining what mix of devices will perform well at minimum cost in the customer's data center. This optimization task is highly complex because of the number of device types available along with the variability of workloads in the data center.

To address this challenge, two things are needed: configuration tools to assist in building such systems and to demonstrate potential benefits based on customer workload, and capabilities in the storage systems that can optimize placement of data in the tiers of storage. The placement should ensure that actively accessed data is co-located to minimize latency while lightly accessed data is placed most economically. There is also an opportunity to improve operating cost by placing data on the minimum set of devices that can serve the workload while powering down the rest. Current products address some but not all of these challenges. Determining which mix of devices to buy remains a difficult problem, and improvement of operating cost by consolidation and power management has not yet been tackled.

To address these gaps, we develop an Extent-based Dynamic Tiering (EDT) system that includes: 1) a Configuration Adviser tool *EDT-CA* to calculate cost-optimized mixes of devices that will service a customer's workload, and 2) a Dynamic Tier Management *EDT-DTM* component that runs in the configured storage system to place data by *dynamically* moving *extents* (fixed-size portions of a volume) to the most suitable tiers given current workload. EDT-CA works by simulating the dynamic placement of extents within tiers that offer the lowest cost to meet an extent's I/O requirements as they change over time, and thus suitably size each tier. EDT-DTM monitors active workload and manages extent placement and migration in such a way that performance goals are met while optimizing operating cost where feasible by consolidating data into fewer devices within each tier and powering off the rest.

We evaluated EDT-CA and EDT-DTM, using both production and synthetic workloads on a storage system with SSDs, SAS, and SATA drives. Our results show that multi-tier systems using EDT have a device mix that saves between 5% to 45% in cost, consume up to 54%

less peak power, and an additional 15-30% lower dynamic power (instantaneous power averaged over time), at a better or comparable performance compared to a homogeneous SAS storage system. EDT's design choices are critical in achieving these savings. Dynamic extent placement saves 25% in cost compared to a static extent-based system. Including metrics in addition to IOPS in EDT's placement provides a 2× performance improvement compared to a dynamic tiering system that allocates extents based on IOPS alone.

Our work makes the following contributions:

- EDT is the first publicly available work that formalizes and explores the design space for storage configuration and dynamic tier management in SSD-based multi-tier systems. (Section 2)
- EDT consists of a novel configuration algorithm for dynamic tiered systems that outputs lower cost configurations. (Sections 3, 4)
- EDT proposes a novel dynamic placement algorithm to satisfy performance requirements while minimizing dynamic power. (Section 5)
- EDT outperforms SAS-only and other simpler extent-based tiering approaches across a variety of workloads in both cost and power. (Section 6)

2 Multi-Tiering: Design Choices

This section describes important design choices for a multi-tier system that enable efficient use of the tiers.

2.1 Extent-based Tiering

The first we consider the granularity of data placement. Previous studies [7, 11] suggest that I/O activity is highly variable across LBAs in a volume. Therefore, if data were placed at a volume level based on average volume workload characteristics, a large percentage of the tier will hold data that does not require the tier's capabilities.

Thus, we perform data placement at the granularity of an **extent**, a fixed-size portion of a volume. The smaller the extent size, the more efficient will be the data placement. However, operating at the extent level incurs metadata overhead to keep track of extent locations and other statistics and this overhead increases as extent size is decreased. We choose an extent size with an acceptable system overhead (details in § 6.2). Note that we expect the extent size to be larger than the typical file system block size and hence extents are not expected to align with file boundaries. However, the reduced system overhead for larger extents provides the right tradeoff compared to finer grain approaches.

2.2 Dynamic Tiering

The next design choice deals with the time scale at which extents move across tiers. One choice involves placing extents once during system instantiation or moving them at coarse grain intervals of the order of days or months. However, since studies show that I/O rates of a workload are typically below peak most of the time [16, 19], this static or semi-static placement is not optimal—a placement that configures for the peaks pays extra in both cost and energy for a system that is over-provisioned at off-peak times; and a placement that mitigates cost from over-provisioning by configuring for the average I/O rate suffers from decreased performance during peaks.

The alternate choice is to plan extent movement at intervals on the order of minutes or hours. We refer to this time interval as an **epoch**. Such a system exploits variation in extent I/O rate to improve its efficiency; an extent is on a SATA tier when fairly inactive, and moves to the SAS or SSD tier as its I/O rate goes up. This achieves cost-effective use of resources and/or dynamic energy savings. Similarly, when the performance demanded of a single tier is below its peak capacity, extents placed on the tier can be consolidated into fewer devices for power savings. Often, the set of heavily loaded extents changes over time [11]. Dynamic migration of the heavily loaded extents into SAS or SSD when required enables cost-effective use of the resources. Thus, we choose to perform dynamic data placement with an epoch length of the order of minutes/hours.

The drawback of such a dynamic system, however, is the cost of data migration, i.e., the potential adverse effect on foreground I/O latency and the migration latency itself before the desired outcome. Longer epoch durations allow more time to execute migrations and amortize overhead better. Thus, we pick an epoch duration whose estimated migration overhead is below the allowable system migration overhead (details in § 6.2). Additionally, it is important to ascertain that the overhead of migrating data does not overwhelm its benefit. This depends on the stability of the workload—extents that relocate often benefit less from migration compared to extents that stay longer in a particular tier. The workloads we have studied indicate that dynamic migration is typically beneficial, but we believe that a dynamic system must also be able to back off when lack of workload stability causes dynamic migration to interfere with performance.

2.3 Beyond I/O Rate Based Tiering

This design choice determines the extent-level statistics required to match an extent with the right tier. The available public documentation about commercial extent-based multi-tier products indicates use of IOPS to measure load; in these systems high IOPS regions are placed onto SSD while leaving the remainder of the data on SAS or SATA. Although this method is intuitively correct, our preliminary analysis reveals significant drawbacks: IOPS-based placement does not factor in the bandwidth

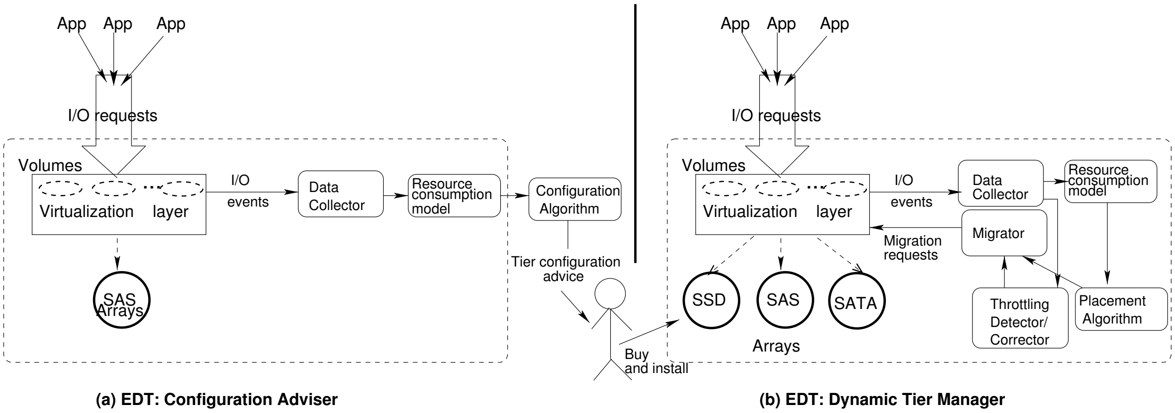


Figure 1: EDT system architecture.

requirement of an extent. For example, consider an extent with a long sequential access pattern consisting of small I/Os to contiguous locations. Such an extent will have high IOPS and bandwidth requirements. Our analysis of production and SPC-1 [1] like workload traces (§ 6), collected after the I/O scheduler show such patterns. Using I/O rate statistics for this stream causes sequential streams, which are more cost-effectively served on SAS or even SATA, to be inappropriately placed on SSD. IOPS placement also ignores capacity of the extent. An extent with high IOPS relative to other extents may not have high enough I/O density (IOPS/GByte) to justify the high \$/GByte cost of the SSD.

Our approach is to collect more than just I/O counts. We employ a heuristic as in [22] to break down an extent's workload: I/Os that access LBAs within 512 KBytes of the previous ones are taken as part of a sequential stream and contribute to an extent's bandwidth requirement. I/Os further apart are characterized as *random* I/Os and are used to compute a random I/O rate. Thus, for each extent, we collect a random I/O rate and bandwidth. Other methods for separating the I/Os into random and sequential may also be applicable.

3 EDT: Design Overview

EDT consists of two elements as depicted in Figure 1: a Configuration Adviser (EDT-CA) that determines the right number of devices per tier to install into a storage system, and a Dynamic Tier Manager (EDT-DTM) that operates inside a running system and continuously manages extent placement across tiers. EDT is expected to be deployed in a commercial storage system as shown in Figure 1 which exports many volumes, includes a virtualization layer that allows volumes to be made up of extents stored in arrays of different device types, is capable of collecting and exporting statistics about extent workloads, and can execute requests to non-disruptively move extents between storage devices.

An example usage scenario is as follows: A user wishes to replace a SAS based storage array with a new, tiered storage system with twice the capability. He collects a trace of his workload over a 24 hour period that he thinks is representative. The trace is then run through EDT-CA which produces the minimum cost configuration of SSD, SAS, and SATA that can provide 2x the performance of the existing system. EDT-CA is aware of the runtime migration capabilities of EDT-DTM and takes them into account when determining the configuration. The user installs the new system. During operation of the new system, EDT-DTM manages migration between tiers by continuously collecting extent level statistics, consolidates data onto lower-power tiers when possible, and monitors the system to ensure that the workload performance is not throttled.

In general, EDT-CA starts by determining the workload requirements for the system it is going to configure. This can either be done with a user generated general description of requirements including IOPS, seq/random mix, length of I/O requests, and their distribution across extents, or by using time series data collected from a workload running on an existing system. For the scope of this work, we assume availability of time series statistics. In this approach, EDT-CA takes a epoch-granularity trace of extent workload statistics sampled at times when storage system usage is high. It then estimates the resources required in different tiers to satisfy that workload by simulating placement of each extent in a tier that minimizes its incurred cost while meeting its performance requirements. It repeats this process every epoch and assigns extents to their lowest cost tier based on their performance requirements in that epoch. At the end of this simulation, EDT-CA determines the set of devices that are needed based on the maximum number of devices needed in each tier over all the epochs. This configuration determines the set of devices purchased by the user.

Once the new tiered system is up and running, EDT-DTM manages extent placement. It collects extent level statistics, estimates extents' resource consumption in different tiers, and then plans and executes migrations. EDT-DTM implements a throttling correction mechanism to ensure that performance requirements are satisfied as they vary over time; it constantly monitors array performance and if performance throttling is detected relocates extents to restore performance. EDT-DTM's placement algorithm seeks to place each extent into the lowest-energy tier that satisfies its performance requirement and then to further minimize energy by consolidating extents in the same tier into fewer devices allowing unused devices to be powered down. Both these algorithms use a Migrator module to move extents.

EDT-CA and EDT-DTM work together to minimize cost. EDT-CA minimizes acquisition cost, and EDT-DTM minimizes operating cost. As our results will show, configurations based on static extent placement are more expensive both to acquire and operate.

3.1 Common Components

EDT-CA and EDT-DTM share components that collect statistics and calculate resource consumption.

3.1.1 Data Collector

The *Data Collector* receives information about I/O completion events including the transfer size, response time, logical block address (LBA) , the volume ID to which the I/O was issued, and the array which executed the I/O. The collector then maps the (LBA, volume id) pair of each I/O to a unique extent in the system, and compiles for each extent, the number of random I/Os and the number of transferred bytes. It then periodically (every minute in our implementation) computes instantaneous bandwidth and random IOPS per extent as well as an exponentially-weighted moving average. In addition to the extent statistics, the collector aggregates statistics per array. It maps each I/O to its array and compiles its IOPS and average response time. These measurements are used by EDT-DTM to determine if I/Os on an array are being throttled. For a very large system the amount of data collected by the data collector may be significant. If this is an issue, the the extent size can be made larger to reduce the volume of statistical data.

3.1.2 Resource Consumption Model

The *Resource Consumption Model* uses the extent statistics to estimate the resources it consumes when placed on a device of a given type. Resources are allocated based on the observed capacity and performance requirements at the device level. Therefore, any workload optimizations like deduplication, compression, and caching do not need to be considered in these models as their effects will be captured by the usage statistics.

An extent consumes the resources of a device along capacity and performance dimensions. Consider an extent of size E_c and a performance requirement E_p determined by its random IOPS rate (*RIOR*) and bandwidth measured in previous epochs. The fraction of capacity required to host an extent E in device D ($RC(E_c,D)$) is straightforward:

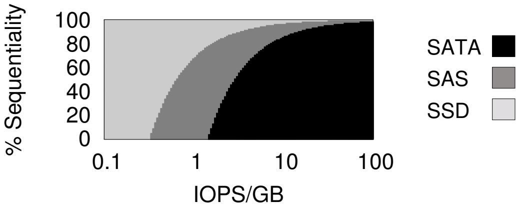


Figure 2: Lowest cost tier for extents with different characteristics.

$$RC(E_c,D) = \frac{\text{Capacity required by extent}}{\text{Total space in device}}$$

For performance utilization, we use a simplified model based on Uysal *et al.*'s work [30]. The performance resource consumption of extent E, when placed on device D ($RC(E_p,D)$) is:

$$RC(E_p,D) = \text{RIOR} \cdot \text{Rtime} + \text{Bandwidth} \cdot \text{Xtime}$$

Here *RIOR* is the number of random I/Os sent to an extent in a second (IO/s) and Rtime is the expected response time of the device (s/IO). *Bandwidth* is the bandwidth requested from the device (MB/s), and *Xtime* is the average transfer time (s/MB). The result of this equation is the fraction of the device performance utilized by an extent. Note that the Rtime and Xtime values are averages and may need to be adjusted depending on the expected workload. For example an SSD with a mostly random write workload would have significantly higher Rtime than the same SSD with a mostly random read workload. The overall resource required by an extent is then the maximum of the capacity utilization fraction and the performance utilization fraction:

$$RC(E,D) = max(RC(E_p,D),RC(E_c,D))$$

The resource consumption model determines the most efficient tier for an extent. For instance, when minimizing cost, the most suitable tier is the one where the extent incurs the lowest cost (the product of the device cost and the extent's resource consumption on that device). Figure 2 confirms the advantage of multi-tier systems since the most cost-effective tier changes with extent characteristics, namely the total IOPS and the percentage of sequential accesses among three classes of storage devices specified in Section 6. As expected, we observe that mostly idle extents favor SATA, medium IOPS favor SAS, and high IOPS favor SSD. Further, as expected, more sequential extents favor HDDs.

4 Configuration Adviser

EDT-CA builds on the Data Collector and the Resource Consumption Model described above. Since configuration is an NP-Hard packing problem, we propose a light-weight heuristic to achieve low cost extent placement:

1. **Binning.** For each extent E, and device type D, we compute the cost of allocating the extent to that device as extent cost(E, D) = cost(D) · RC(E,D). The extent is then placed in the tier that meets its performance with the lowest cost. Iterating over all the extents, the above computation separates the extents into bins, one per each tier.
2. **Sizing a bin.** For each bin, we obtain its performance and capacity resource consumption as $RC_p = \sum RC(E_p, D)\ \forall E$, and $RC_c = \sum RC(E_c, D)\ \forall E$. The maximum of these two values gives the total bin resources required, and the number of required devices of this bin type are computed by rounding up this sum to the nearest integer value.
3. This process is independently repeated for each epoch to identify the number of devices per tier that yields minimum cost for that epoch.
4. The last step consists of combining these different configurations to obtain a final system configuration valid across time. For the scope of this work, we achieve the final configuration by allocating the maximum number of devices of each type used across all epochs. That is, if at epoch t_0 2 devices of type D and 1 of type D' are the most cost effective, but at epoch t_1 1 of type D and 2 of D' is better, then our method will indicate that we need 2 of type D and 2 of D'.

Our current method of combining configurations across epochs is fairly conservative and could potentially result in an over-provisioned system. However, as our current algorithm already results in lower cost configurations (Section 6), we relegate exploring more efficient ways of combining configurations over time to future work. Also note that when we compute tiered configuration for each epoch independently, we assume that the extents can be suitably migrated between epochs if required. As part of our future work, we intend to model the required number of migrations, and suitably adjust the provisioning if the required migrations exceed the maximum number of migrations a system can support in a chosen interval of time. Finally, our Configuration Algorithm can also be used to upgrade a multi-tier system to meet upcoming performance demands.

5 Dynamic Tier Manager

EDT-DTM combines three new modules with the Data Collector and the Resource Consumption Model to continuously optimize extent placement: (1) a Tiering and Consolidation module, (2) a Throttling Detector/Corrector module, and (3) a Migrator module.

5.1 Tiering and Consolidation Algorithms

At the end of every epoch, the Tiering and Consolidation (TAC) algorithms generate an extent placement to satisfy extent performance requirements and minimize dynamic system power. Such an energy efficient placement can be achieved both by leveraging the strengths (i.e. performance or capacity per watt) of the heterogeneous underlying hardware (SSD, SAS, and SATA drives), and by consolidating data into fewer devices when possible and turning off the unused devices.

Similar to the configuration problem, placement for power minimization is also NP-Hard, and we propose a heuristic solution. TAC requires two inputs: (1) current random I/O rate and bandwidth for each extent from the actively running system, and (2) size (in bytes) and the random I/O rate and bandwidth capability for each array in the storage system. It then uses a two-step process to output a new extent placement that aims to adapt to the changes in the workload as follows:

(1) **Tiering.** For each extent E, and device type D, we compute the "fractional power burden" of allocating the extent to that device as extent power(E, D) = power(D) · RC(E,D). The extent is then placed on the tier that meets its performance with the lowest power consumption. Doing so allows EDT to reduce active power via consolidation (described next). Iterating over all the extents results in one bin per tier. The assignment of extents to a tier is performed locally on an extent by extent basis, irrespective of the total performance needs or available space in that tier.

(2) **Consolidation.** Extents assigned to each tier are then sorted using their *RC* values and placed in arrays using the First Fit Decreasing heuristic, a good approximation algorithm to the optimal solution for extent packing [35]. When extents already assigned to the tier under consideration exceed its available performance (i.e., resource consumption metric for the assigned extents exceeds 1) or the tier runs out of space in the available arrays, the remaining extents in the extent list are demoted to the tier with the next lower power burden for that extent. This packing process is now repeated for all the tiers, consolidating extents into a minimum number of arrays in a tier. Extents already in the right tier and on an array that will remain powered on in this epoch retain their position from the previous epoch, thereby saving migrations. Any unused arrays from the extent placement are set to a lower power state to conserve energy.

5.2 Throttling Detector and Corrector

While the TAC mechanisms enable dynamic performance and power optimization, unexpected load and working set changes can suddenly alter the performance requirements of extents. However, tracking this performance change, especially when an extent's I/O rate increases, is challenging. Extents placed in a low performance tier cannot exhibit high I/O rates even when the application above may desire it. This causes *throttling* of the true IOPS requirement of the extent, artificially limiting it to a low value. The Throttling Detector overcomes this limitation by monitoring the average response time of each active array every minute.

If the average response time of I/Os from an array indicates that undesirably high request queuing is occurring in the array, EDT decides that the array is throttling the true IOPS requirement of applications and causing delays. When throttling is detected, pending migrations driven by TAC are immediately halted and EDT-DTM switches to a *throttling correction* mode to perform recovery. To respond rapidly and minimize the possibility of future throttling in the same array, the load on the throttled array is shed by migrating a minimum set of extents responsible for at least half of its current total performance resource consumption.

To select the target array(s), we first start by considering the best possible tier for each extent being migrated, and within that tier we first examine arrays which are already active to see if they can absorb the new extent. If none can host the new extent, we consider arrays that are not in use in that tier if any are available. If the best tier can not accommodate the extent we try the same approach on tiers with the next higher power burden for that extent. If the array continues to remain throttled after half the load on the array has been migrated, the extent migration process is repeated, until the system is no longer throttled. The entire system stays in recovery mode while an array remains throttled, suspending energy optimizing migrations. When no arrays are throttled, the system switches back to the TAC placement after an epoch elapses.

5.3 Migrator

The Migrator handles the data movement requests from TAC and the throttling algorithms. It compares the new placement of the extents from the above algorithms to their old placement, and identifies extents that need to be migrated. It then schedules and optimizes these migrations. On one hand, migrations that relieve throttling must be completed quickly. On the other hand, migrations cause additional I/O traffic, and care must be taken so that they do not affect the foreground I/O performance.

Our migration scheme achieves this tradeoff as follows. We allow every device to be involved in only one migration operation at a time. Thus, before issuing a migration request, the Migrator performs admission control by allowing requests only if the source and target device are both available. If they are not, the request is re-queued and it moves onto the next request. Further, the Migrator controls its migration-related resource consumption by decomposing an extent into smaller transfer units and *pacing* the transfer requests to match the minimum of the available or the desired I/O rate. Further if the migration is being performed to relieve throttling, once a transfer unit is migrated, any foreground I/O requests to it are handed by the destination array. Note that because of this pacing not all planned migrations may be completed before the next epoch. In such cases, the migration queue is flushed, and requests resulting from the new epoch's computation are queued. We further optimize by retaining the old location of the extent if it is already in the right tier during the consolidation step. Finally, we could potentially incorporate other optimizations [4, 9, 31, 36] such as multiple locations for the same extent [31], and proactive migrations [36].

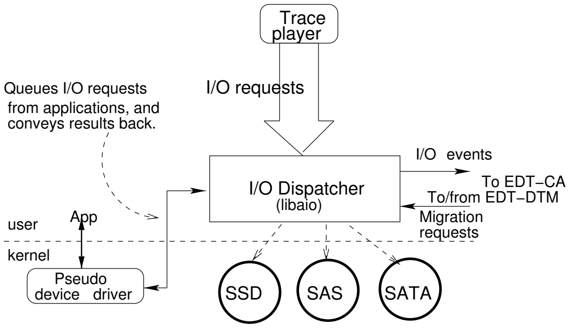


Figure 3: Storage subsystem platform for evaluating EDT-CA and EDT-DTM.

6 Evaluation

Our evaluation uses both a SPC-1-like [1] benchmark workload and multiple production enterprise workloads from MSR [21] to demonstrate that:

- In comparative evaluation, EDT-CA works to minimize cost, and EDT-DTM satisfies performance requirements while lowering power consumption.
- EDT's dynamic behavior and detailed resource consumption model help achieve its goal.
- Extent based dynamic optimization and consolidation are feasible in practice with little overhead.

6.1 Methodology

Comparison candidates. We compare EDT to three alternate solutions:

1. *SAS* is chosen to represent current enterprise storage system deployments that predominantly use only high performance SAS drives. The configuration is derived

Device	Cost ($)	Power (Idle, Active)	Random IOPS	BW (MB/s)	Rtime (ms/IO)	Xtime (ms/KB)
SSD	430	0.5, 1	5000	90	0.2	0.01
SAS	325	12.4, 17.3	290	200	3.75	0.004
SATA	170	8.0, 11.6	135	105	9	0.009

Table 1: Characteristics of devices used in the testbed.

using the capacity and **peak** performance (IOPS and bandwidth) requirements of the workload. Volumes are statically assigned to SAS arrays in a load-balanced manner.

2. *EST* (Extent-based Static Tiering) places extents on tiers statically to quantify the benefit from tiering. Configuration is performed as follows: at every epoch, the cost to place each extent on each tier is computed as done by EDT-CA using capacity, IOPS, and bandwidth requirements. An extent is then permanently placed on the tier that minimizes the sum of its instantaneous costs over all epochs. Once extents are binned into tiers, the number of devices for each tier is determined using that tier's peak resource consumption.

3. While SAS and EST illustrate the benefit from EDT's design choices incrementally (going from a homogeneous system to static tiering and then to dynamic tiering), we propose a third candidate to illustrate a different design decision in dynamic multi-tier systems—*IDT* (IOPS Dynamic Tiering) implements extent-based dynamic configuration and placement using a greedy IOPS-only criteria where higher IOPS extents move to higher IOPS tiers. This is in contrast to EDT that uses a combination of capacity, IOPS, and bandwidth in its placement algorithm.

Implementation. Our test system is shown in Figure 3. In addition to EDT, we implemented an I/O dispatcher that receives block I/O requests from applications, maps the logical block address to the physical device address, performs the corresponding I/Os, and communicates with the EDT components. Our *trace player* application issues block I/Os from a trace via a socket to the I/O dispatcher. To support real-world applications without modification, we implemented a pseudo block device interface. For the scope of this work, we use Linux's default *deadline* scheduler, and our measurement of context switch overhead when running through the pseudo device driver was negligible ($< 10\mu$s).

Experimental Testbed. Our experimental platform consists of an IBM x3650 with 4 Intel Xeon cores and 4 GB memory acting as the I/O dispatcher. It is connected via internal and external SAS ports to 12 1 TB 7200 rpm 3.5" SATA drives, 12 450 GB 15K rpm 3.5" SAS drives, and 4 180 GB Intel X25-M SSD drives. Table 1 shows the characteristics of these devices. The enclosures containing the drives are connected to a *Watts up? Pro* power meter. We report the disk power obtained by subtracting the baseline power used by the non-disk components of the enclosure (154 W).

Metrics. To compare solutions, we evaluate static configuration results using capital cost and peak power consumption, and we evaluate dynamic behavior using the average and distribution of I/O latency along with dynamic power consumption. Peak power consumption is obtained using disk drive data sheets. Dynamic power consumption is measured using the power meter.

6.2 Parameter Selection

Extent size. Smaller extents use tier and migration-related resources more efficiently and enable faster response to workload changes, but also incur greater metadata overhead. Our approach was to pick the smallest extent size that incurs acceptable metadata overhead. Assuming metadata can have a reasonably small overhead of at most 0.0001% of the total storage capacity, and given 200 bytes/extent for metadata overhead (mostly from recording extent-level statistics) in our implementation, the smallest extent size our storage system can support is 20 MB. To introduce some slack we used 64 MB extents for our experiments.

Epoch duration. Shorter epochs allow quicker response to workload changes, but can also result in increased extent migration. As the epoch duration increases, the stability of extent characteristics increases due to averaging over longer periods and consequently the migration bandwidth overhead decreases. We picked epoch durations that resulted in migration bandwidth limited to a 10% fraction of the available array-pair bandwidth in the system[1]. This prevents migration from significantly degrading performance and ensures that migrations complete early within each epoch. For the MSR workloads this calculation resulted in a 30 minute epoch.

6.3 Synthetic Workload

This SPC1-like workload was chosen because it simulates an industry standard benchmark and provides a contrast to the MSR trace workloads. We ran the SPC1-like workload generator on a 1 TB volume at 100 BSUs for 30 min using an over-provisioned configuration (a 12 SAS RAID-0 array). We chose 30 min because the workload is quite static after a short startup period. The resulting trace was used to obtain the number of devices required per tier for different methods (Table 2).

We observe that all the extent-based tiering configurations outperform SAS configurations in both capital cost and peak power consumption. EDT reduces cost by 14%, and peak power by 55% compared to SAS. Cost incurred

[1] Medium to large scale tiered storage systems would typically perform simultaneous extent migrations across multiple array-pairs.

System	# of Disks	Energy	Cost	Avg RT
SAS	(0, 6, 0)	103.8 W	$1950	28 ms
EST	(2, 2, 1)	46.6 W	$1680	15 ms
IDT	(2, 1, 1)	29.3 W	$1355	21 ms
EDT	(2, 2, 1)	46.6 W	$1680	15 ms

Table 2: Configuration for synthetic workload. The number of disks per tier is specified as (SSD, SAS, SATA). The average response time is obtained from running the configuration with 100 BSUs .

to configure EST and EDT for this relatively static workload are similar. Although the IDT configuration seems to provide the least cost configuration, this is an artifact of rounding up required devices to the next higher integer. Using fractional devices, costs for EDT and IDT are much closer ($890 vs. $920). Note that in larger systems rounding effects will be less significant.

To confirm that EDT's lower cost is not at the expense of performance, we ran the SPC1-like workload for 30 minutes at 100 BSUs. Given the stability of the workload, migration overhead was minimal. We therefore chose an epoch of 5 minutes to complete the experiments quickly. The SAS scheme used 6 SAS RAID-0 array. Other schemes operated on individual disks. We started EDT and IDT with the entire volume in the SATA tier and allowed dynamic extent migration to reach optimal configurations over time. EST, which does not support extent migration, was started with extents in their most suitable locations as per the EST configuration.

The last column of Table 2 shows the average response times for 100 BSUs measured starting at the end of the first epoch, once the extent placements of the dynamic tiering configurations become effective. Given the workload's stability, results for EDT and EST are identical. They both achieve a 40% lower response time compared to SAS, and improve on IDT's IOPS only placement by 20%. Note that the dynamic power consumption in these experiments is similar to the peak power due to the lack of workload variation.

6.4 Production Workload

Our next workload (*MSR-combined*) represents the more interesting class of real-world workloads, obtained by combining the I/Os to the 31 (out of 36) most active volumes of a production storage system [21] for a total of 4580 GB. Including the remaining 5 volumes was not feasible given the hardware restrictions of our testbed.

Configuration outcomes. Configuration outcomes based on six days of the *MSR-combined* workload, shown as the "Equal Performance" group in Table 3, indicate that the tiering configurations have lower cost compared to SAS. EDT incurs the lowest cost (50% reduction compared to SAS and 25% relative to EST). EDT's ability to effectively time share high-cost, high-performance tiers across extents and satisfy sequentially accessed ones with the SAS tier (instead of the SSD tier) results in more cost-effective configurations. Extents placed in the SATA tier (4336 GB) are mostly idle with random IOPS below 0.32, those in SAS (69 GB) are dominated by bandwidth higher than 1.45 MB/s and random IOPS less than 1.43, and the SSD extents (175 GB) have random IOPS between 1.45 and 858. Tiered configurations substantially reduce peak power when compared with SAS; IDTs greater use of the SSD tier (relative to SAS) makes it the most power-efficient.

Config	System	# of Disks	Energy	Cost
Equal Performance	*SAS*	(0, 16, 0)	276.8 W	$5200
	EST	(5, 2, 4)	82 W	$3480
	IDT	(4, 1, 4)	64.5 W	$2725
	EDT	(3, 2, 4)	81.6 W	$2620
Equal Cost	*SAS*	(0, 12, 0)	204 W	$3900
	EST	(4, 4, 4)	116 W	$3700
	IDT	(4, 4, 4)	116 W	$3700
	EDT	(4, 4, 4)	116 W	$3700

Table 3: Configuration for *MSR-combined*. Configurations achieving equal performance depict improvement in cost and peak power. Configurations at equal cost are created for experimental ease. Number of disks in each tier specified as (SSD, SAS, SATA).

Performance and Power outcomes. Not all of the equal performance configurations listed in Table 3 were feasible on our experimental testbed due to hardware limitations. Consequently, we decided to switch to equal cost configurations (shown in Table 3) to contrast performance at equal cost instead of cost at equal performance only for the *MSR-combined* workload. Later, we shall explore equal performance configurations for feasible subsets of volumes (Figure 6). EDT's configuration was chosen as the base for all the tiering systems, and its configuration requirements were rounded up to integer number of arrays, each array consisting of 4 devices. SAS used only SAS drives for the same cost, split into 4 disk RAID 0 arrays. We then replayed day one from the seven day trace, the most active 24 hour period of the *MSR-combined* workload. Both EDT and IDT were bootstrapped using a load balanced volume placement.

Figure 4 summarizes the results of this experiment for the candidate solutions. First, we notice that the I/O response time distribution of EDT is clearly superior to the other three solutions, highlighting the importance of considering random IOPS, bandwidth, and capacity when making tiering choices. The average response time with EDT was 2.94 ms while those for the SAS, EST, and IDT were 5.12, 9.33, and 5.93 ms respectively. Further, the 95^{th} percentile response time for EDT was under 7.86

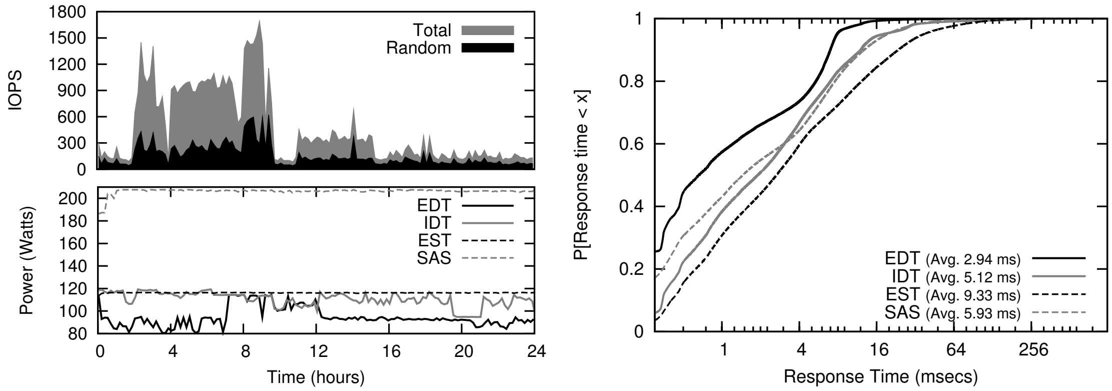


Figure 4: I/O rate and power consumption (left) and response time distribution (right) for *MSR-combined*.

ms while the same for SAS, EST, and IDT were 19.31, 37.06, and 17.891 ms respectively. On average, EDT decreased the dynamic power consumption by 13% relative to its peak power, 55% relative to SAS and at least 10% relative to IDT and EST. This dynamic power savings result is likely to underestimate power savings observed in real deployments given that the workload was generated by consolidating multiple uncorrelated workload traces, which tended to reduce the workload variability that would enable dynamic power savings. Additionally, the experiment was done over the most active period, which required most devices to be active for performance. Further, all the configurations here are sized to meet the observed workload. Typically, however, storage purchases are made to accommodate future growth and hence over-provisioned to begin with, resulting in more dynamic power savings.

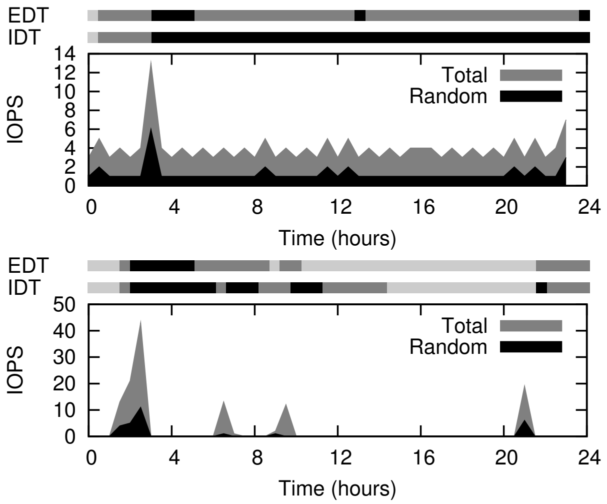


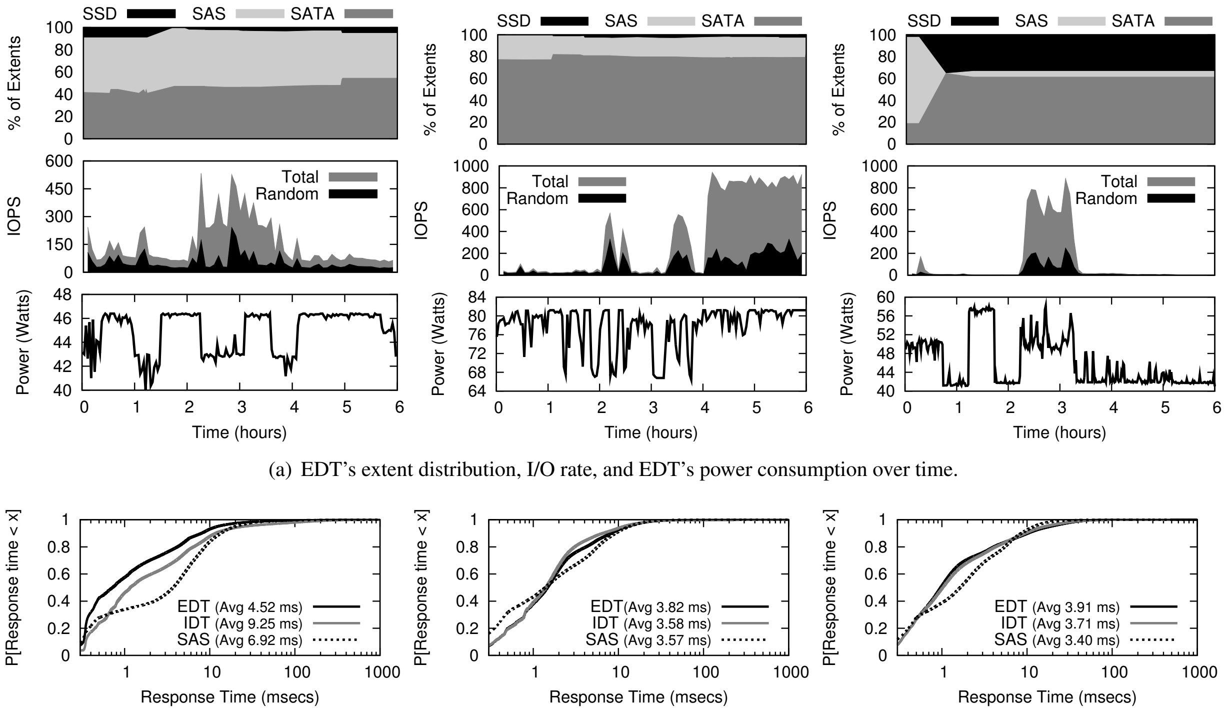
Figure 5: Contrasting extent migrations for EDT and IDT. The two upper lines denote extent placement for the different algorithms. Black is SSD tier, dark grey SAS and light grey SATA.

Analysis. We illustrate how EDT achieves its superior performance using two example extents chosen from the experiment and contrasting them with IDT. Figure 5 shows the sequential and random IOPS over time for two extents along with the tier they are placed in. For extent A (top graph), both IDT and EDT move the extent from the SATA tier (the default initial location) to higher performing tiers when the total IOPS requirements increase. However, IDT allocates the SSD tier starting from hour 3 on account of the exponentially weighted moving average (EWMA) of total IOPS whereas EDT allocates the SSD tier only when the EWMA of *random* IOPS of the extent is high. Thus, EDT can better capitalize on the superior sequential performance of the SAS tier to minimize capital costs during configuration and sustain performance during operation. Extent B (bottom graph) illustrates similar behavior during predominantly sequential accesses. Further, both EDT and IDT rightly move extent B into the SATA tier when it becomes idle, aiding in power savings. Thus, EDT is successfully able to pick the best tier for an extent's workload and relocate it when the requirements change. Regarding the overheads for this migrations, both EDT and IDT migrated around 120 extents per epoch, using an average bandwidth of 42 MB/s which only represents 3% of the total available.

Workload	Volumes	Cap (GB)	Accessed
server	hm, mds, prn, prxy, stg, ts, wdev, web	1650	30%
data	proj, rsch, usr	3719	34%
srccntl	src1, src2	904	29%

Table 4: Sub-workloads derived from MSR.

Varying the workload. To analyze the sensitivity of the various algorithms to workload characteristics, we grouped volumes from the MSR workload as specified in Table 4 to create the *server*, *data* and *srccntl (source*



(a) EDT's extent distribution, I/O rate, and EDT's power consumption over time.

(b) CDF of response times

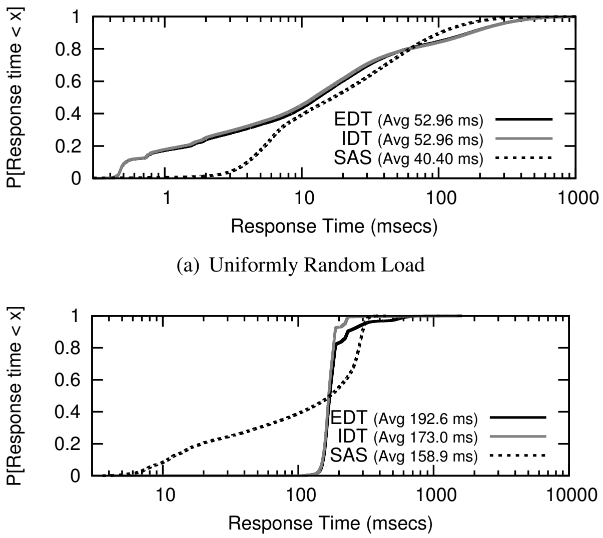
Figure 6: Replaying 6 hours of the MSR sub-workloads. First column is *server*, second *data*, and third *srccntl*.

Workload	System	# of Disks	Energy	Cost
server	*SAS*	(0, 6, 0)	103.8 W	$1950
	EST	(2, 1, 2)	42.5 W	$1525
	IDT	(2, 1, 1)	30.9 W	$1355
	EDT	(1, 2, 1)	47.2 W	$1250
data	*SAS*	(0, 10, 0)	173 W	$3250
	EST	(2, 2, 3)	71.4 W	$2020
	IDT	(1, 2, 4)	82 W	$1760
	EDT	(1, 2, 4)	82 W	$1760
srccntl	*SAS*	(0, 6, 0)	103.8 W	$1950
	EST	(2, 3, 1)	65.5 W	$2005
	IDT	(2, 2, 2)	59.8 W	$1850
	EDT	(2, 2, 2)	59.8 W	$1850

Table 5: Configuration for MSR sub-workloads. Number of disks in each tier specified as (SSD, SAS, SATA).

code control) workloads. Configuration outcomes for each sub-workload using SAS, IDT, and EDT are presented in Table 5. As with *MSR-combined*, the dynamic tiering solutions are able to configure both lower-cost and lower-energy systems when compared with SAS and EST. Further, in the case of the *server* workload, EDT optimizes the configured system cost with a single SSD relative to the two SSDs recommended using IDT. Given that EST had significantly inferior performance for *MSR-combined*, we did not consider it for further analysis.

Figure 6 shows EDT's dynamic power consumption and extent distribution across tiers over time, as well as its response time distribution relative to IDT and SAS. First, unlike *MSR-combined*, these workloads do have substantial periods of lower utilization. Consequently, in addition to improving the capital cost and peak power consumption, EDT's dynamic consolidation allows dynamic power savings of as much as 15-31% relative to its peak power across the three workloads. The extent distribution is quite different across the workloads. EDT uses the SSD tier substantially for the *srccntl* workload. IOPS-wise one would think that the workload should be completely consolidated to the SATA; however, EDT leverages the fact that the SSD tier offers improved energy efficiency for up to 40% of the extents. The SAS tier was most used for *server*, in particular between hours 2-4 when sequential activity dominates. The *data* workload predominantly utilizes the SATA tier (as evidenced in the configuration outcome) since the IOPS per extent for most extents is very low, easily accommodated using SATA devices. Finally, in this equal performance configuration experiment, the response time performance with EDT is either similar or better than the SAS and IDT schemes across the workloads.



(a) Uniformly Random Load

(b) Varying Hot Extent Set

Figure 7: Extent distribution and CDF for the adversarial workload.

6.5 Adversarial Workloads

Finally, we measure the impact of using EDT with workloads completely different than the one it is provisioned for. We used the configuration obtained for the *srccntl* workload (in Table 5), and instead of the trace from that workload, we ran two separate synthetic workloads for two hours each: (1) a uniformly random workload at 400 IOPS, where each I/O is issued to a random page in the system. (2) a workload at 500 IOPS, where I/Os are issued to a chosen set of 10 hot extents initially in the SATA tier and this set changes every minute.

Figure 7 depicts the distribution of response times for both workloads. The uniformly random workload yields a 31% higher average response time for EDT and IDT compared to SAS. This can be attributed to the constant migration I/O moving extents away from the throttled SATA tier to both SAS and SSD tiers. Interestingly, we see only a 21% penalty for EDT in the second workload. Analysis shows that throttling of the newly active extents was promptly detected and the extents were migrated quickly to the SSD before they became cold. As illustrated by these examples, EDT can handle unexpected workloads using its throttling detection/correction techniques without major performance penalties.

7 Related Work

We build on a rich body of related work in multiple areas.

SSD-based storage architectures. Several products (IBM's EasyTier [29], EMC's FAST [17], 3PAR [25], and Compellent [23] systems) incorporate SSDs in storage tiering solutions. Since technical details of these approaches are not published, EDT is the first to provide insight into design choices and components, detailed evaluation across workloads, and analysis of benefits and challenges in building SSD-based multi-tier systems. Moreover, the publicly available documents of these products indicate that although they achieve cost savings and performance improvements, there is little focus on tools aiding admins/customers to configure the right device mix for their workload or on incorporating algorithms that target dynamic energy savings. EDT addresses these limitations.

Another approach to leverage solid state technology in storage systems is to deploy flash devices as a cache between DRAM and HDD. NetApp's FlashCache [24] which follows this approach cites cost reduction and performance improvement when coupled with SAS/SATA drives. Interestingly, Narayanan *et al.* [22] have argued that a SSD cache layer above SAS disks was generally not cost effective compared to an all SAS configuration at the same performance. We did find cost savings using SSD, but our system included much lower cost SATA disks to improve overall cost. Unfortunately, a detailed comparison between SSD caching and tiering would take a significant effort and more space than is available in this paper. However, our summary thoughts on the two architectures are: 1) SSD caching will utilize the SSD space more efficiently and can be more responsive to very dynamically changing workloads, but 2) SSD tiering enables both cost and energy savings even in enterprise environments.

Storage configuration (also referred to as *provisioning*). Systems such as Minerva [3], Hippodrome [5], and DAD [6] address the problem of optimizing storage configuration by iteratively applying several steps such as configuring a low cost storage system, choosing RAID levels and other array parameters, and assigning entire volumes to arrays. EDT-CA's focus on obtaining the right mix of storage devices to minimize cost is similar to the configuration step in these systems. The key difference is that EDT-CA is inherently aware of, and utilizes the flexibility afforded by EDT's dynamic extent placement. EDT's data layout also operates at a much finer extent granularity. In EDT, we use a model to predict the utilization of an extent (given its bandwidth and random IOPS) that is similar in spirit to the previously proposed store level performance predictor [30] in its accounting for the differential load induced by sequential and random accesses to an extent. Finally, EDT-CA can be enhanced to perform utility based provisioning as in [28].

Tiering. Migration-based storage tiering has been prevalent in the industry for a long time in the form of Hierarchical Storage Management systems, Information Lifecycle Management solutions, and other forms of coarse-grain tiering [2, 13, 15]. Most of these systems differ from EDT in that they generally migrate data from upper to lower tiers, based on its age rather than on load. Further, these systems operate on volume, file system, or file objects rather than extents, and as such are suited more for file layer systems than block layer systems. Wilkes *et al.* propose AutoRAID [33], a storage system where extents within volumes are migrated between faster RAID-1 arrays and slower RAID-5 arrays according to workload and age. Significantly different algorithms for migration decisions tuned to the specific two tiers are proposed. Additionally, AutoRAID does not consider the issue of correctly determining a device mixture to satisfy given workloads.

Storage energy efficiency. EDT uses a consolidation algorithm to save energy in primary storage systems. Other energy saving approaches that instead spin down a fraction of the available disk drives with active data [8, 10, 18, 20, 21, 26, 27, 31, 32, 34] either are not applicable in many primary storage systems due to the significant spin up latency, or require undesirable capacity over-provisioning for redundant data. Work leveraging Dynamic RPM capability (e.g., [12, 26, 37, 38]). is complementary to EDT. In fact, Hibernator [38] also leverages tiering but varies RPM setting of the drives to minimize energy.

8 Discussion

Extending the resource consumption model In this work we assumed RAID-0 arrays when estimating how much resource on a tier is consumed by a given workload. In commercial applications of EDT, more sophisticated models will be needed to estimate resource consumption in arrays with different RAID levels. Such models do already exist in the industry, so we believe incorporating this capability will be straightforward. Also, for the scope of this work, we assume that all arrays are at the same reliability level, and hence migrating data across arrays is not restricted. However, it is feasible to remove this constraint by observing policies to limit the migration targets of extents. Finally, the resource model may need be enhanced to better model the behavior of disks servicing multiple sequential IO streams in parallel. The current model does not account for degradation in sequential performance that may occur when a disk needs to service multiple sequential streams at once.

Disk power fraction in the overall energy of a storage system. The chief dynamic energy-saving technique proposed in this work is powering down empty disk drives. However, we find that in today's commercial storage systems, disk drives typically consume ~50% of the total storage system energy [14] while the rest is consumed by other components which do not currently have the capability of varying their energy consumption according to workload. As these components overcome this limitation, our energy-saving techniques can be extended to include them, leading to a more energy proportional system and lower overall operating costs.

Applicability. The target domain for EDT is primary storage systems where response time is critical. Archival applications where response time is not as critical may be better served with existing solutions using policy-based migration and power-saving storage such as spun-down disk or tape. Also, EDT will be most effective when the working set and I/O intensity are somewhat stable with some variation. When the workload is static, dynamic migration will not take place but consolidation will still be beneficial if the system is not capacity bound.

9 Conclusion

The increasing availability of solid-state drives has ushered in a new era of multi-tiered primary storage systems. With EDT, we have formalized the *configuration* and *dynamic tier management* problems and have systematically explored the design choices available when building such systems. We presented the design, implementation, and evaluation of EDT's Configuration Adviser (EDT-CA) and Dynamic Tier Manager (EDT-DTM). EDT lowers capital cost by configuring less expensive tiered storage and operating costs by dynamically optimizing power consumption via consolidation whenever feasible. We also demonstrated that EDT is successfully able to address the data migration overheads of dynamic tiering and respond rapidly and effectively to unexpected changes in the workload.

Experimental results show EDT has significant benefit. Evaluation performed using both a production workload and industry-standard synthetic workload revealed that multi-tier systems using EDT have a device mix that saves between 5% to 45% in cost, consume up to 54% less peak power, and an additional 15-30% lower dynamic power (instantaneous power averaged over time), at a better or comparable performance compared to a homogeneous SAS storage system. Experimental results also demonstrated that EDT is superior to simpler alternatives for extent-based tiering, providing lower cost and better performance, and consuming similar or lesser power. We hope that this study serves as a starting point for future work along the promising direction of multi-tiered enterprise storage systems.

Acknowledgments

We would like to thank our shepherd Hakim Weatherspoon, our anonymous reviewers, and Renu Tewari, Aameek Singh and Amar Phanishayee for their valuable feedback. This work was supported in part by NSF grants CNS-0747038 and CNS-1018262. Jorge Guerra was supported in part by an IBM PhD Fellowship.

References

[1] SPC specifications. http://www.storageperformance.org/specs.

[2] M. K. Aguilera, K. Keeton, A. Merchant, K.-K. Muniswamy-Reddy, and M. Uysal. Improving recoverability in multi-tier storage systems. In *Proc. of the IEEE/IFIP DSN*, 2007.

[3] G. A. Alvarez, E. Borowsky, S. Go, T. H. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A. Veitch, and J. Wilkes. Minerva: An Automated Resource Provisioning Tool for Large-scale Storage Systems. *ACM Transactions on Computer Systems*, 19(4):483–518, 2001.

[4] E. Anderson, J. Hall, J. Hartline, M. Hobbs, A. R. Karlin, J. Saia, R. Swaminathan, and J. Wilkes. An experimental study of data migration algorithms. *Lecture Notes in Computer Science*, 2141/2001: 145–158, 2001.

[5] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: Running Circles Around Storage Administration. In *Proc. of USENIX FAST*, 2002.

[6] E. Anderson, S. Spence, R. Swaminathan, M. Kallahalla, and Q. Wang. Quickly finding near-optimal storage designs. *ACM Transactions on Computer Systems*, 23(4):337–374, 2005.

[7] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis. BORG: Block-reORGanization for Self-Optimizing Storage Systems. *Proc. of USENIX FAST*, 2009.

[8] D. Colarelli and D. Grunwald. Massive Arrays of Idle Disks for Storage Archives. In *Proc. of IEEE/ACM SC*, 2002.

[9] K. Dasgupta, S. Ghosal, R. Jain, U. Sharma, and A. Verma. Qosmig: Adaptive rate-controlled migration of bulk data in storage systems. In *Proc. of ICDE*, 2005.

[10] K. M. Greenan, D. D. Long, E. L. Miller, T. J. Schwarz, and J. J. Wylie. A Spin-Up Saved is Energy Earned: Achieving Power-Efficient, Erasure-Coded Storage. In *Proc. of USENIX HotDep*, 2008.

[11] J. Guerra, H. Pucha, K. Gupta, W. Belluomini, and J. Glider. Energy Proportionality for Storage: Impact and Feasibility. In *Proc. of ACM/USENIX HotStorage*, 2009.

[12] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Frankez. DRPM: Dynamic speed control for power management in server class disks. In *Proc. of ACM/IEEE ISCA*, 2003.

[13] IBM Corporation. High Performance Storage System (HPSS). Online: http://hpss-collaboration.org/, 2010.

[14] IBM Corporation. IBM System Storage DS8000 series. Data Sheet, 2010.

[15] G. Karche, M. Mamidi, and P. Massiglia. Using dynamic storage tiering. Available as Symantec Yellow Books at http://www.symantec.com/enterprise/yellowbooks/index.jsp., 2006.

[16] R. Koller and R. Rangaswami. I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance. In *Proc. of USENIX FAST*, 2010.

[17] B. Laliberte. Automate and Optimize a Tiered Storage Environment FAST! ESG White Paper, 2009.

[18] H. J. Lee, K. H. Lee, and S. H. Noh. Augmenting RAID with an SSD for Energy Relief. In *Proc. of USENIX HotPower*, 2008.

[19] A. Leung, S. Pasupathy, G. Goodson, and E. Miller. Measurement and Analysis of Large-Scale Network File System Workloads. In *Proc. of USENIX ATC*, 2008.

[20] D. Li and J. Wang. EERAID: Energy efficient redundant and inexpensive disk array. In *Proc. of workshop on ACM SIGOPS European workshop*, 2004.

[21] D. Narayanan, A. Donnelly, and A. Rowstron. Write Off-Loading: Practical Power Management for Enterprise Storage. In *Proc. of USENIX FAST*, 2008.

[22] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron. Migrating Server Storage to SSDs: Analysis of Tradeoffs. In *Proc. of ACM Eurosys*, 2009.

[23] M. Peters. Compellent harnessing ssds potential. ESG Storage Systems Brief, 2009.

[24] M. Peters. Netapp's solid state hierarchy. ESG White Paper, 2009.

[25] M. Peters. 3par: Optimizing io service levels. ESG White Paper, 2010.

[26] E. Pinheiro and R. Bianchini. Energy conservation techniques for disk array-based servers. In *Proc. of ACM ICS*, 2004.

[27] E. Pinheiro, R. Bianchini, and C. Dubnicki. Exploiting redundancy to conserve energy in storage systems. *SIGMETRICS*, 34(1), 2006.

[28] J. D. Strunk, E. Thereska, C. Faloutsos, and G. R. Ganger. Using utility to provision storage systems. In *Proc. of USENIX FAST*, 2008.

[29] Taneja Group Technology Analysts. The State of the Core Engineering the Enterprise Storage Infrastructure with the IBM DS8000. White Paper, 2010.

[30] M. Uysal, G. A. Alvarez, and A. Merchant. A modular, analytical throughput model for modern disk arrays. In *Proc. of IEEE MASCOTS*, 2001.

[31] A. Verma, R. Koller, L. Useche, and R. Rangaswami. SRCMap: Energy Proportional Storage Using Dynamic Consolidation. In *Proc. of USENIX FAST*, 2010.

[32] C. Weddle, M. Oldham, J. Qian, A.-I. A. Wang, P. Reiher, and G. Kuenning. PARAID: A Gear-Shifting Power-Aware RAID. In *Proc. of USENIX FAST*, 2007.

[33] J. Wilkes, R. Golding, C. Staeliin, and T. Sullivan. The HP AutoRAID Hierarchical Storage System. In *Proc. of ACM SOSP*, 1995.

[34] X. Yao and J. Wang. RIMAC: a novel redundancy-based hierarchical cache architecture for energy efficient, high performance storage systems. *SIGOPS Operating Systems Review*, 40(4), 2006.

[35] M. Yue. A simple proof of the inequality ffd(l) (11/9)opt(l) + 1, for all l, for the ffd bin-packing algorithm. *Acta Mathematicae Applicatae Sinica*, 7:321331, 1991.

[36] G. Zhang, L. Chiu, C. Dickey, L. Liu, P. Muench, and S. Seshadri. Automated Lookahead Data Migration in SSD-enabled Multi-tiered Storage Systems. In *IEEE MSST*, 2010.

[37] 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 *Proc. of IEEE HPCA*, 2004.

[38] Q. Zhu, Z. Chen, L. Tan, Y. Zhou, K. Keeton, and J. Wilkes. Hibernator: helping disk arrays sleep through the winter. In *Proc. of ACM SOSP*, 2005.